SUMMER PROCEEDINGS 2015

The Center for Computing Research at Sandia National Laboratories

Editors:
Andrew M. Bradley and Michael L. Parks
Sandia National Laboratories

December 18, 2015

SAND#: SAND2016-0830 R

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energys National Nuclear Security Administration under contract DE-AC04-94AL85000.

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

Printed in the United States of America. This report has been reproduced directly from the best available copy.

Available to DOE and DOE contractors from

U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone:	(865) 576-8401
Facsimile:	(865) 576-5728
E-Mail:	reports@adonis.osti.gov
Online ordering:	http://www.doe.gov/bridge

Available to the public from

U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone:	(800) 553-6847
Facsimile:	(703) 605-6900
E-Mail:	orders@ntis.fedworld.gov
Online ordering:	http://www.ntis.gov/ordering.htm

Preface

The Center for Computing Research (CCR) at Sandia National Laboratories organizes a summer student program each summer, in coordination with the Computer Science Research Institute (CSRI) and Cyber Engineering Research Institute (CERI).

CERI focuses on open, exploratory research in cyber security in partnership with academia, industry, and government, and provides collaborators an accessible portal to Sandia's cybersecurity experts and facilities. Moreover, CERI provides an environment for visionary, threat-informed research on national cyber challenges.

CSRI brings university faculty and students to Sandia National Laboratories for focused collaborative research on DOE computer and computational science problems. CSRI provides a mechanism by which university researchers learn about problems in computer and computational science at DOE Laboratories. Participants conduct leading–edge research, interact with scientists and engineers at the laboratories, and help transfer the results of their research to programs at the labs.

A key component of CCR programs over the last decade has been an active and productive summer program in which students from around the country conduct internships at Sandia. Each student is paired with a Sandia staff member who serves as technical advisor and mentor. The goals of the summer program are to expose the students to research in the mathematical and computer sciences at Sandia and to conduct a meaningful and impactful summer research project with their Sandia mentor. Every effort is made to align summer projects with the student's research objectives and all work is coordinated with the ongoing research activities of the Sandia mentor in alignment with Sandia technical thrusts.

Starting in 2014, CERI and CSRI combined their efforts to form the CCR Summer Proceedings. Both CERI and CSRI encourage all summer participants and their mentors to contribute a technical article to the CCR Summer Proceedings. In many cases, the CCR Proceedings are the first opportunity that students have to write a research article. Not only do these proceedings serve to document the research conducted during the summer but, as part of the research training goals of Sandia, it is the intent that these articles serve as precursors to or first drafts of articles that could be submitted to peer–reviewed journals. As such, each article has been reviewed by a Sandia staff member knowledgeable in that technical area with feedback provided to the authors. Several articles have or are in the process of being submitted to peer–reviewed conferences or journals and we anticipate that additional submissions will be forthcoming.

For the 2015 CCR Proceedings, research articles have been organized into the following broad technical focus areas—*computational mathematics, applications, and software and high performance computing*—which are well aligned with Sandia's strategic thrusts in computer and information sciences.

We would like to thank all participants who have contributed to the outstanding technical accomplishments of CSRI and CERI in 2015. The success of the program hinged on the hard work of enthusiastic student collaborators and their dedicated Sandia technical staff mentors. We would also like to thank those who reviewed articles for this proceedings; their feedback is an important part of the research training process and has significantly improved the quality of the reports.

An important educational component of the summer program is the CCR Summer Seminar Series. We would like to thank the staff who spoke at the 2015 Series: Laura Matzen (org. 1463), Jason Wheeler (6533), Bruce Hendrickson (1400), Jean-Paul Watson (1464), Carter Edwards (1426), Megan Slinkard (5752), Mark Taylor (1446), Stephen Olivier (1423), Steve Plimpton (1444), and Ojas Parekh (1464).

Finally, the CCR summer program would not be possible without the administrative and IT support of Ashley Avallone, Jonathan Compton, Steven Garcia, Bill Goldman, Denise LaPorte, Amy Levan, Lorena Martinez, Sandra Portlock, Phyllis Rutka, and Bernadette Watts.

Andrew M. Bradley
Michael L. Parks

December 18, 2015

Table of Contents

Computational Mathematics

Computational mathematics is concerned with the design, analysis, and implementation of algorithms to solve mathematical problems. Articles in this section describe the discretization of partial differential equations and methods to solve equations, couple multiphysics systems of equations, and quantify uncertainty in models.

Ashcraft, Niederhaus, and Robinson verify the convergence speed and the accuracy of a new formulation of the discretization of the two-dimensional cylindrical symmetric magnetic diffusion equations. As part of this work, they develop a new manufactured solution for their method of manufactured solution study. The verified method can now replace the less efficient one in ALEGRA.

Cheung, Perego, and Bochev use alternating Schwarz methods to address two problems of coupling. The first problem is that of gaps in coupled interfaces, as for example those that occur when two contacting objects are meshed separately. The second application couples Density Functional Theory to the Poisson-Nernst-Planck equations.

Conde and Shadid find implicit-explicit (IMEX) Runge–Kutta schemes for the additive initial value problem that are strong stability preserving (SSP) in the explicit and implicit parts separately and also in the overall scheme. The schemes are demonstrated on a convection-diffusion equation and coupled first-order wave equations.

Costa, Bond, Littlewood, and Moore describe a nonlocal multiscale finite element method (MSFEM) that solves the peridynamic model at multiple spatial scales. The MSFEM is an efficient means to incorporate small-scale heterogeneities into a coarse-scale solution.

Garavito-Garzón and Lehoucq perform numerical experiments to determine the robustness, sensitivity, and dependence on boundary conditions of a functional used in a PDE-constrained optimization approach to estimating the velocity field in image data.

Glaws, Wildey, and Shadid explore active subspaces, a new dimension reduction method. They summarize the method; analyze methods to estimate the dimension reduction error; apply these methods to standard verification problems in computational fluid dynamics and magnetohydrodynamics to provide a framework for understanding the relationship between the active variables and the underlying physics; and then apply them to a more challenging problem.

Hajghassem, Cyr, and Ridzal develop methods that parallelize the time dimension of optimization problems constrained by linear time-dependent PDE. Their method is based on developing a parallel-in-time preconditioner.

Jamal and Turner focus on speeding up multiscale computational methods. They apply an efficient coupling schedule to recursively subdivided spatial subdomains in a peridynamics application to couple subdomains in time and space at different time scales. The schedule is computed using subdomain-specific models of computational cost.

Lopez and Taylor study existing, and propose new, limiters for the spectral element core of the Community Atmosphere Model. They use a simplified chemistry model and a test problem that has sharp tracer gradients to determine the extent to which each limiter maintains linear tracer correlations.

Mátyásfalvi, Eckstein, and Watson implement an object-oriented, parallel optimizer called FOAAL. It has interfaces to AMPL and PYOMO. They apply it to a unit commitment problem.

Moe, Bochev, Peterson, and Ridzal develop a new optimization-based remap (OBR) algorithm suited to incremental remap and semi-Lagrangian transport applications. In this OBR method, global mass conservation is enforced while approximately minimizing the subset of the grid on which the update has a nonphysical domain of dependence. Results of

this method are compared with a previous method that did not attempt to minimize this subset.

Mundt, Boslough, Taylor, and Roesler create a more accurate dual grid for use in the climate science High-Order Methods Modeling Environment (HOMME). The dual grid is important for the conservation of energy and water in climate simulations. They analyze the performance of remapping operations using their dual grid.

Patel, Boman, Rajamanickam, and Chow describe the implementation of a fine grained, asynchronous, iterative algorithm to compute incomplete approximate factorization preconditioners for sparse matrices. The algorithm reformulates the factorization process as the iterative solution of a nonlinear system of equations by a fixed point method. Approximate application of the preconditioner is essentially a sequence of matrix-vector products and so also is largely fine grained and asynchronous.

A.M. Bradley
M.L. Parks

December 18, 2015

VERIFICATION OF AN IMPROVED CYLINDRICAL MAGNETIC DIFFUSION ALGORITHM IN ALEGRA

CLAYTON C. ASHCRAFT*, JOHN H. NIEDERHAUS†, AND ALLEN C. ROBINSON‡

Abstract. ALEGRA is one of Sandia's simulation codes used to simulate magnetohydrodynamic phenomena. One application of this code is to calculate the magnetic flux density, B_θ, in a continuum by solving the two-dimensional cylindrical symmetric magnetic diffusion equations; however, ALEGRA's current B_θ formulations do not converge optimally. We seek to implement a new formulation that converges both quickly and accurately by implementing a change in coordinates inspired by a paper written by J.B.M. Melissen and J. Simkin [2]. We do so, and perform a detailed verification analysis. Our verification study uses a wire-in-a-void model to create a non-trivial test problem for which we can derive an analytical solution. We derive one using perturbation theory, but find it is non-uniformly accurate near $t = 0$ and is less than satisfactory for completing our verification study. We formulate a new analytical solution using the method of manufactured solutions, and are then able to show that the new formulation performs as expected. With the new formulation verified, we briefly discuss a few additional results and some research on an exploding wire problem using the new formulation. Details omitted in this paper will be available in a future Sandia report[1].

1. Introduction. The ALEGRA code was written to conveniently solve the resistive magnetohydrodynamic (MHD) equations, which are frequently used to model high-current, high-energy field phenomena in a conducting continuum. Operator splitting in ALEGRA results in a magnetic diffusion sub-problem of the form:

$$\frac{\partial \mathbf{B}}{\partial t} = -\nabla \times (\frac{1}{\sigma}(\nabla \times \frac{\mathbf{B}}{\mu})) \tag{1.1}$$

or

$$\sigma \left(-\frac{\partial \mathbf{A}}{\partial t} - \nabla \phi \right) = \nabla \times (\frac{\nabla \times \mathbf{A}}{\mu}) \tag{1.2}$$

where σ is the electrical conductivity, $\mathbf{B}$ is the magnetic flux, and $\mathbf{A}$ is the vector potential. In this paper, we are interested in the two dimensional simplifications of Equation 1.1 when the problem is axi-symmetric and the fundamental variable is the out-of-plane magnetic field component, B_θ.

Code has already been developed in ALEGRA to find the value of B_θ under these conditions with the constraint that the magnetic permeability, μ, is constant. If a non-constant μ is necessary or desired, one would need to solve using an H_θ formulation, which we do not discuss here.

*Brigham Young University, cc.ash@byu.edu
†Sandia National Laboratories, jhniede@sandia.gov
‡Sandia National Laboratories, acrobin@sandia.gov

We seek to improve the B_θ code. We show that the original, straightforward Fully Integrated Finite Element (FIFE) formulation of the code, which we will simply refer to as the FIFE formulation from now on, can converge very slowly to the correct solution, making it difficult to justify the computational expense required to obtain useful simulation data. An alternately implemented formulation, which we will call the R-Scaled formulation, solves for rB_θ and converges much more rapidly, but it tends to be inaccurate near $r = 0$, reducing its utility on problems which require accurate computation in that region. Both of these issues restrict the researcher's ability to analyze problems of interest, so here we introduce a new formulation, referred to as the PSI-S formulation, to rectify these issues and make ALEGRA even more robust and capable.

The PSI-S formulation was inspired by a paper written by J.B.M. Melissen and J. Simkin [2]. In their paper, they apply a clever change of coordinates to the cylindrical coordinate version of Equation 1.2, resulting in the elimination of a singularity in the spatial discretization and a large overall improvement in the discrete error. We use the same strategy with Equation 1.1 to produce the PSI-S formulation. Eliminating the singularity generates an equation set perfectly suited for linear finite elements.

Thus, we expect the PSI-S formulation to be more accurate and to converge faster than the current formulations, and in order to demonstrate this superiority, we perform a verification analysis and report the results below.

2. Finite Element Formulation. Under 2D cylindrically symmetric assumptions, ALEGRA performs computations in an (r, z) coordinate system, where r represents the radius in cylindrical geometry, and z represents the orthogonal direction in the plane. Following Melissen and Simkin, we perform a coordinate change on the R-Scaled formulation, taking it from (r, z) coordinates to (s, z), where $s = r^2$ and setting $\psi = rB_\theta$ to be the dependent variable. This relatively simple change eliminates a singularity in the finite element magnetic diffusion equations used by ALEGRA and is the foundation of our entire report.

In this section, we will give the finite element magnetic diffusion equation for the new formulation, along with its matrix equivalent. Some derivation is included, but most will be skipped for the sake of maintaining brevity in this report. More detail can be found in the future Sandia report[1].

2.1. Magnetic Diffusion Equation. We begin with the the magnetic diffusion equation that ALEGRA solves after the Lagrangian hydrodynamic motion step, given by Equation 1.1, with μ set equal to μ_0, the magnetic permeability of free space. Moving to to 2D cylindrical geometry and using the magnetic flux function $\psi = rB_\theta$, this becomes

$$\frac{\partial \psi}{\partial t} = r\frac{\partial}{\partial r}\left(\frac{\eta}{\mu_0 r}\frac{\partial \psi}{\partial r}\right) + \frac{\partial}{\partial z}\left(\frac{\eta}{\mu_0}\frac{\partial \psi}{\partial z}\right). \tag{2.1}$$

where $\eta = \frac{1}{\sigma}$. Noting that the $\frac{1}{r}$ coefficient presents a problem as $r \to 0$, we make the change in coordinates: $s = r^2$, and $ds = 2rdr$. The resulting magnetic diffusion equation is:

$$\frac{1}{s}\frac{\partial \psi}{\partial t} = \frac{\partial}{\partial s}\left(\frac{4\eta}{\mu_0}\frac{\partial \psi}{\partial s}\right) + \frac{\partial}{\partial z}\left(\frac{\eta}{s\mu_0}\frac{\partial \psi}{\partial z}\right). \tag{2.2}$$

Note that now the s does not appear inside an s derivative.

2.2. Matrix Equation for Magnetic Diffusion. Continuing from Equation 2.2, we multiply by an arbitrary test function W, and take the volume integral of 2.2 over a domain or sub-domain, Ω, resulting in Equation 2.3.

$$\begin{aligned}
&\int_\Omega W \frac{1}{s}\frac{\partial \psi}{\partial t} dsdz \\
&= \int_\Omega W \left(\frac{\partial}{\partial s}\left(\frac{4\eta}{\mu_0}\frac{\partial \psi}{\partial s}\right) + \frac{\partial}{\partial z}\left(\frac{\eta}{s\mu_0}\frac{\partial \psi}{\partial z}\right)\right) dsdz \\
&= \int_\Omega W \left(\nabla \cdot M(s) \nabla \psi dsdz\right) \qquad (2.3)
\end{aligned}$$

where

$$M(s) = \begin{bmatrix} \frac{4\eta}{\mu_0} & 0 \\ 0 & \frac{\eta}{\mu_0 s} \end{bmatrix}.$$

Applying the identity $\nabla \cdot (f\mathbf{A}) = (\nabla f) \cdot \mathbf{A} + f(\nabla \cdot \mathbf{A})$, the divergence theorem, and Ohm's Law, we arrive at

$$\begin{aligned}
&\int_\Omega W \frac{1}{s}\frac{\partial \psi}{\partial t} dsdz + \int_\Omega \frac{\partial W}{\partial s}\frac{4\eta}{\mu_0}\frac{\partial \psi}{\partial s} + \frac{\partial W}{\partial z}\frac{\eta}{\mu_0 s}\frac{\partial \psi}{\partial z} dsdz \\
&= \int_{\partial\Omega} W (S\mathbf{E}) \cdot \mathbf{t} dl \qquad (2.4)
\end{aligned}$$

where $\mathbf{t}$ is the tangent unit vector, $S = \begin{bmatrix} \frac{1}{\sqrt{s}} & 0 \\ 0 & 2 \end{bmatrix}$, $\mathbf{E}$ is the electric field in the plane, and l is the arc-length in the (s, z) plane.

Note the presence of the $1/s$ terms in the integrals. While these terms might initially appear to be problematic as $s \to 0$, the solution and the test function are required to be $O(s)$ as $s \to 0$, so the integrals are not singular.

Finally, using the normal linear finite element assembly process where we set $\psi = \sum_j \psi_j N_j$, N_j being the standard linear finite element basic functions, and applying a backwards Euler time discretization, we get the elemental matrices

$$\begin{aligned}
M^e_{ij} &= \int_{\Omega_e} N_i \frac{1}{s} N_j dsdz \\
K^e_{ij} &= \int_{\Omega_e} \left(\frac{\partial N_i}{\partial s}\frac{4\eta}{\mu_0}\frac{\partial N_j}{\partial s} + \frac{\partial N_i}{\partial z}\frac{\eta}{\mu_0 s}\frac{\partial N_j}{\partial z}\right) dsdz \\
L^e_i &= \int_{\partial\Omega_e} N_i (S\mathbf{E}) \cdot \mathbf{t} dl
\end{aligned}$$

which assemble to the global equations

$$M_{ij}\frac{\psi_j^{n+1} - \psi_j^n}{\Delta t} + K_{ij}\psi_j^{n+1} = L_i^{n+1/2}. \qquad (2.5)$$

3. Verification. In order to perform our verification analysis, we need to design a test problem that is simple enough for analytical study, yet has all the features of typical MHD problems of interest. In this case, problems of interest are those which have large jumps in material conductivity near the axis, have the magnetic field out of the plane, and are described by the magnetic flux density diffusion equations. These types of problems are common, and the various B_θ formulations are applicable.

Our problem of choice is a wire in a void model, with a constant, axial tangential electric field assumed to exist along the interface. Figure 3.1 shows the problem's physical set up: A cylindrical rod of height h, radius a, and conductivity σ_r, surrounded by a void region of conductivity $\sigma_v \ll \sigma_r$, and depth $b - a$. We assume an instantaneous jump at $t = 0$ to a constant, axial-tangential electric field along the interface of the rod and the void, which generates a time dependent magnetic flux, B_θ, through the rod and the void.

This magnetic flux is well understood, and has a solvable analytic solution. Inside the rod, the solution for B_θ with respect to the rod radius should appear as a diffusion wave into the rod early in time until reaching a radial linear steady state profile. This will happen regardless of the dimensions of the rod or the magnitude of the electric field applied, and a complete analytical expression for this behavior is attainable using Laplace transforms. In the void, Ampere's law says that B_θ should behave approximately like $\frac{\mu_0 I(t)}{2\pi r}$ for all time, where $I(t)$ is the total current enclosed and r is the distance from the z axis. However, this is merely the leading order solution. We also want to demonstrate a robust solution methodology when the conductivity in the void is non-zero. The B_θ formulation is unable to handle zero-valued conductivity, so we set the void conductivity to be very small relative to the conductivity of the rod (e.g. $\frac{\sigma_v}{\sigma_r} \approx 10^{-6}$). The ramification is that now the solution in the void region is perturbed, making the analytical solution more difficult to derive. We could still, in principle, solve the inner-outer region exact solution via transform techniques, but this is unnecessary, and would not provide much value; so instead, we choose the electric field we desire at the interface $r = a$, for which we have an exact solution, and then compute the electric field boundary condition for the interface $r = b$ that will impose it. This will provide an excellent verification of ALEGRA's ability to perform over a conductivity jump.

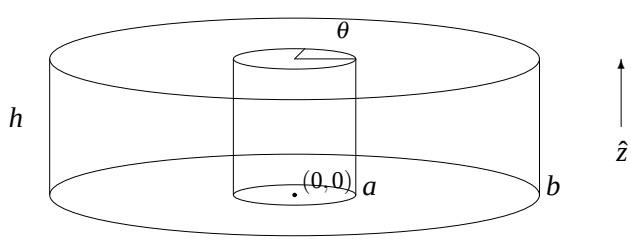


Fig. 3.1: Metal Rod in a Void

3.1. Solution in the Rod. Our strategy for obtaining an analytical solution for this problem is to focus first on the solution in the rod, which we can solve for using Laplace transforms and residue theory, then obtain a perturbation solution in the void that coincides with the solution in the rod at the conductivity jump. Combining the two gives us a complete analytical solution for the test problem that can be used to verify the accuracy, and superiority, of our new numerical formulation. The solution for B_θ in the rod is

$$B_\theta(r,t) = \frac{\mu_0 \sigma_r V}{h}\left(\frac{r}{2} - 2a\sum_{n=1}^{\infty}\frac{e^{-\left(\frac{k_n^2}{\mu_0\sigma_r a^2}\right)t} J_0'\left(\frac{k_n r}{a}\right)}{k_n^2 J_0'(k_n)}\right) \tag{3.1}$$

where V is the voltage drop across the wire, J_0 is the Bessel function of the first kind of order 0, and k_n is the nth root of J_0. As a check, notice that as $t \to \infty$, B_θ goes to a linear profile, which is what we expected.

3.2. Solution in the Void: Perturbation Solution. We now need to approximate a solution in the void region which is consistent with this inner solution for non-zero σ_v. A perturbation analysis in small σ_v is our first approach. Assuming continuity of the electrical and magnetic fields at $r = a$ gives us the solution:

$$B_\theta(r,t) = \frac{\mu_0 I(t)}{2\pi r} + \mu_0\sigma_v\left[\frac{\mu_0 \dot{I}(t)}{2\pi}\left(\frac{r}{2}\ln\left(\frac{r}{a}\right) - \frac{r}{4} + \frac{a^2}{4r}\right) + \frac{V}{2h}\left(r - \frac{a^2}{r}\right)\right]. \tag{3.2}$$

It follows that the boundary condition that we will use in ALEGRA to produce our desired electric field at $r = a$ is an electric field at $r = b$ given by:

$$E(b,t) = \frac{V}{h} + \frac{\mu_0 \dot{I}(t)}{2\pi}\ln\left(\frac{b}{a}\right) + \mu_0\sigma_v\left[\frac{\mu_0 \ddot{I}(t)}{2\pi}\left(\frac{b^2+a^2}{4}\ln\left(\frac{b}{a}\right) - \frac{b^2}{4} + \frac{a^2}{4}\right)\right] \tag{3.3}$$

where, again, $I(t)$ is the total current in the rod, and is given by $I(t) = \frac{2\pi a}{\mu_0} B_\theta(a,t)$.

3.3. Set Up in ALEGRA. We set up our model as a 1mm long cylindrical rod with a 2mm radius, a 3mm deep void (see Figure 3.3), and a $1000V/m$ electric field maintained on the surface of the rod for $t > 0$. We use Equation 3.3 to produce a boundary condition at $r = b$ that should produce the same effect in ALEGRA as if we applied the electric field directly to the surface of the rod. For the electrical conductivity of the rod we choose $\sigma_r = 1\ \mathrm{M}\Omega \cdot m$, and for the electrical conductivity of the void we chose $\sigma_r = 1.0\ \Omega \cdot m$.

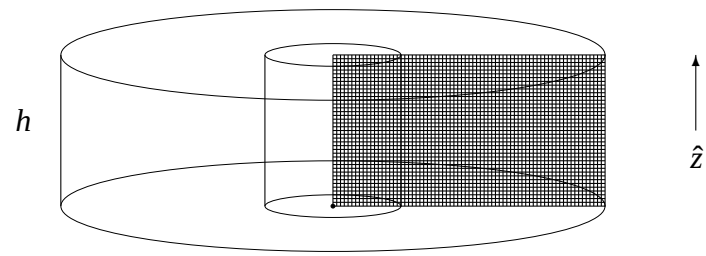


Fig. 3.2: ALEGRA 2D (r,z) Cross Section

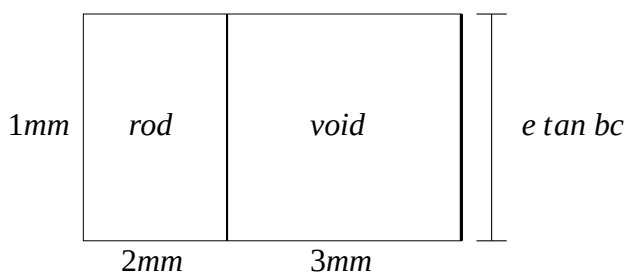


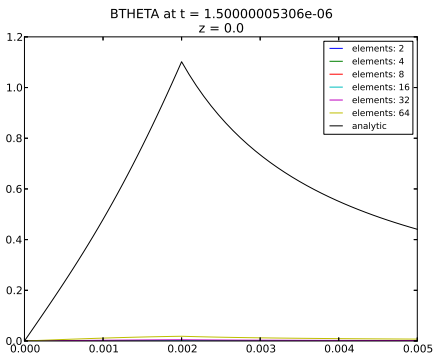
Fig. 3.3: 2D (r,z) mesh with BC on right side

For our mesh, we create two blocks, one representing the rod and another representing the void. This allows us to ensure that the conductivity jump is on the boundaries of elements regardless of how many elements we want to use. We set each block to have only one element in the z direction, and allow the number of elements in the r or s direction to vary. Because we anticipate the PSI-S formulation to converge very quickly, we set the first mesh size in our study to have one element per block in the r or s direction. Thus, our very first mesh has exactly two elements. The strategy we employ for increasing the mesh size is to double the number of elements per block, per test.

In order to get ALEGRA to produce easily interpreted results, we impose a condition on the time steps ALEGRA uses to perform its calculations. The theoretical leading order error in ALEGRA's discrete solution is $O(\Delta t) + O((\Delta r)^2)$, where Δt is the time step used in the calculation, and Δr (or Δs) is the length of a mesh element. So by forcing $\Delta t \propto (\Delta r)^2$, we make our error $O((\Delta r)^2)+ O((\Delta r)^2) = O((\Delta r)^2)$, which allows us to confidently expect an order 2 convergence rate for these tests.

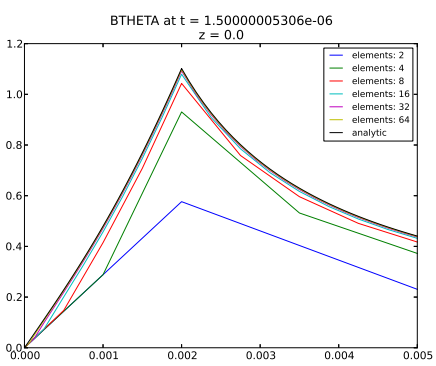
Finally, in order to see whether or not the PSI-S formulation is outperforming previously implemented formulations, we will run ALEGRA on our test problem using the FIFE, the R-Scaled, and PSI-S formulations and then compare.

3.4. Results from Perturbation Solution. Figures 3.4, 3.5, 3.6, show plots of ALEGRA results for the different formulations for the different mesh sizes as well as a table showing the convergence rates measured for each mesh size. The black line represents the analytical solution we derived previously. Each result is given at time $t = 1.5\mu s$, which is fairly representative of our general results and is still in the transient phase of problem. Also, each plot represents nodal data from nodes located at the bottom of the mesh ($z = 0.0$). We could look at the nodal data at the top of the mesh or the elemental data at the center of the mesh as well, but nodal values are more accurate, and, since we have essentially made this problem one dimensional by having only one element in the z direction, we are able to accurately represent the overall results by only examining this set of nodes.



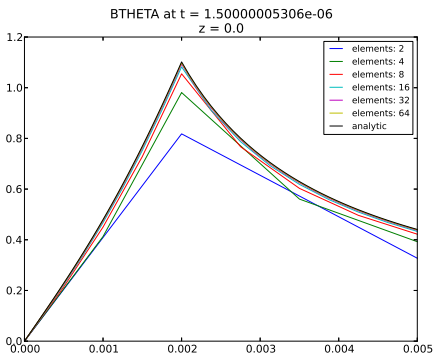
Number of elements per block	Order of convergence
1	2.09611e-01
2	2.09611e-01
4	7.31077e-02
8	2.16658e-02
16	1.01580e-02
32	2.09721e-02

Fig. 3.4: FIFE formulation at $t = 1.5\mu s$



Number of elements per block	Order of convergence
1	1.52564e+00
2	1.52564e+00
4	1.43740e+00
8	1.35715e+00
16	1.29348e+00
32	1.22582e+00

Fig. 3.5: R-Scaled formulation at $t = 1.5\mu s$



Number of elements per block	Order of convergence
1	1.40072e+00
2	1.40072e+00
4	1.38177e+00
8	1.27804e+00
16	1.16535e+00
32	1.07729e+00

Fig. 3.6: PSI-S formulation at $t = 1.5\mu s$

We can see that FIFE formulation does not converge at all for these mesh resolutions (the largest two mesh resolutions are barely distinguishable at the bottom of the plot). As a matter of fact, we do not see convergence for this formulation at all until we have mesh resolutions of thousands of elements. This slow overall convergence is what makes this formulation impractical.

The R-Scaled formulation does better, but seems to be converging at an order one convergence instead of two. We see the same with the PSI-S formulation. This is not what we expected.

To check our work, we construct two variations of our test, and briefly state our results and conclusions. In the first variation, we perform the same test but with the void block removed. We observe second order convergence for the FIFE and PSI-S formulations, and less than second order convergence for R-Scaled formulation. For the second, instead of driving the problem with an electric field boundary condition, we use a magnetic field boundary condition proportional to total current. Here we again observe second order convergence for FIFE and PSI-S, and less than second order convergence for R-Scaled. We attribute the slower convergence rate of the R-Scaled formulation to its inaccuracies near $r = 0$; then, based on these observations, we conclude that the numerical algorithms in ALEGRA are coded correctly.

3.5. Solution in the Void: Manufactured Solution. Because it seems that there are no problems with ALEGRA, we conclude that the problem is the approximate analytical solution. We can see that Equations 3.2 and 3.3 both depend on derivatives of $I(t)$, which are singular at $t = 0$, so some inaccuracies can be expected from these equations. In this case, it seems that these inaccuracies are significant enough to disrupt our verification analysis, particularly the boundary condtion given by Equation 3.3. The remedy, then, is a new analytical solution without these types of singularities.

The method we now employ for finding an analytical solution is the Method of Manufactured Solutions. We assume that our analytical solution for B_θ in the rod holds, and use it to manufacture a solution for B_θ in the void by again matching continuous magnetic and electric fields to our derived interior solution at the interface $r = a$. The solution in the void is

$$B_\theta(r,t) = \frac{\mu_0 \sigma_v V}{2h} r - \frac{c_1 \mu_0}{r} - \sum_{n=1}^{\infty} \lambda_n^2 e^{-\frac{\lambda_n^2}{\mu_0 \sigma_v} t} \left[a_n J_0'(\lambda_n^2 r) + b_n Y_0'(\lambda_n^2 r) \right] \tag{3.4}$$

where Y_0 is the Bessel function of the second kind of order 0, and c_1, λ_n, a_n, and b_n are:

$$\begin{aligned}
c_1 &= \frac{(\sigma_v - \sigma_r)V}{2h} a^2 \\
\lambda_n &= \sqrt{\frac{\sigma_v k_n^2}{\sigma_r a^2}} \\
a_n &= -\left(\frac{2(\sigma_r)^2 \mu_0 a^3 V}{\sigma_v k_n^4 h} \right) \frac{Y_0(\xi_n)}{J_0(\xi_n) Y_0'(\xi_n) - Y_0(\xi_n) J_0'(\xi_n)} \\
b_n &= \left(\frac{2(\sigma_r)^2 \mu_0 a^3 V}{\sigma_v k_n^4 h} \right) \frac{J_0(\xi_n)}{J_0(\xi_n) Y_0'(\xi_n) - Y_0(\xi_n) J_0'(\xi_n)} \\
\xi_n &= \frac{\sigma_v k_n^2}{\sigma_r a}.
\end{aligned}$$

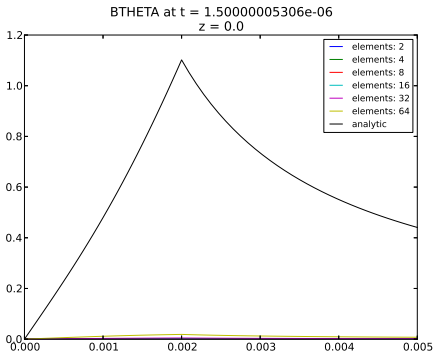
With this solution, we now have to impose both a boundary condition at $r = a$ and an initial condition in ALEGRA. The B_θ initial condition we impose is given by

$$B_\theta(r,0) = \begin{cases} \frac{\mu_0 \sigma_r V}{h} \left[\frac{r}{2} - 2a \sum_{n=1}^{\infty} \frac{J_0'(\frac{k_n r}{a})}{k_n^2 J_0'(k_n)} \right], & r \le a \\ \frac{\mu_0 \sigma_v V}{2h} r - \frac{\mu_0 c_1}{r} - \sum_{n=1}^{\infty} \lambda_n^2 \left[a_n J_0'(\lambda_n^2 r) + b_n Y_0'(\lambda_n^2 r) \right], & r > a \end{cases} \tag{3.5}$$

and the electric field boundary condition is

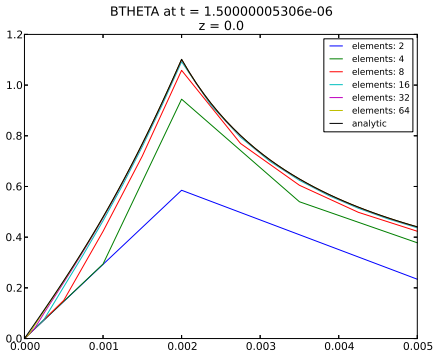
$$E(b,t) = \frac{V}{h} + \frac{1}{\mu_0 \sigma_v} \sum_{n=1}^{\infty} \lambda_n^2 e^{-\frac{\lambda_n^2}{\mu_0 \sigma_v} t} \left[a_n J_0(\lambda_n^2 b) + b_n Y_0(\lambda_n^2 b) \right]. \tag{3.6}$$

3.5.1. Results from Manufactured Solution. We analyze and plot the results of our test with the manufactured solution the same way we did previously, and this time they meet our expectations. As with the perturbation solution, Figure 3.7 shows that the FIFE formulation does not converge at these resolutions. Thousands of elements are needed. However, at resolutions of 100 to 3200 elements, we can show that this formulation is actually second order convergent, but this need for highly resolved meshes to get convergence still makes this formulation impractical to use. Figure 3.8 shows that the R-Scaled formulation is approaching a convergence rate of just under 1.5, which we learned from our simplified problems is to be expected. Finally, Figure 3.9 demonstrates the excellent convergence and accuracy of the PSI-S formulation, successfully concluding our verification analysis.



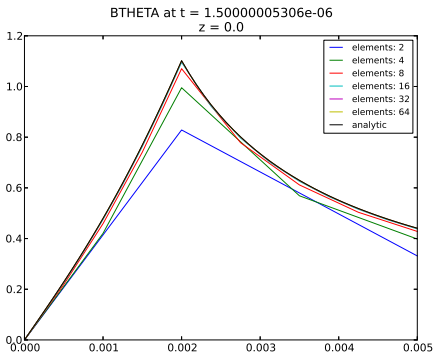
Number of elements per block	Order of convergence
1	2.09611e-01
2	2.09611e-01
4	7.31078e-02
8	2.16662e-02
16	1.01587e-02
32	2.09737e-02

Fig. 3.7: FIFE formulation with manufactured solution at $t = 1.5\mu s$



Number of elements per block	Order of convergence
1	1.59109e+00
2	1.59109e+00
4	1.60632e+00
8	1.66119e+00
16	1.53120e+00
32	1.45542e+00

Fig. 3.8: R-Scaled formulation with manufactured solution at $t = 1.5\mu s$



Number of elements per block	Order of convergence
1	1.52672e+00
2	1.52672e+00
4	1.75725e+00
8	2.24243e+00
16	2.33500e+00
32	2.03655e+00

Fig. 3.9: PSI-S formulation with manufactured solution at $t = 1.5\mu s$

4. Additional Work. In addition to our verification study, we are able to identify several positive effects of the implementation of the PSI-S formulation. We find that the PSI-S formulation improves the accuracy not only of the magnetic flux (B_θ) computations, but also the of current density ($\mathbf{J}$), the pressure, and temperature computations. The differences are significant and clearly visible near $r = 0$.

The effects of the PSI-S formulation on the exploding wire problem, however, are perhaps even more dramatic. We still notice the more accurate computations near $r = 0$; but in addition, we see a considerable variation in wire exploding time and behavior when using the PSI-S formulation compared to the R-Scaled formulation. This suggests that our work may be of significance for many practical problems as well as advancing our original goal of improving solution quality of axi-symmetric momentum diffusion problems in ALEGRA [1].

5. Conclusions. ALEGRA is one of Sandia's simulation codes used to simulate magnetohydrodynamic phenomena. One application of this code is to calculate the magnetic flux, B_θ, in a continuum by solving the two-dimensional cylindrical symmetric magnetic diffusion equations. We have shown that ALEGRA's current B_θ formulations converge non-optimally and sought to develop a new formulation that converged both quickly and accurately. By implementing a change in coordinates inspired by a paper written by J.B.M. Melissen and J. Simkin, we did so, and performed a detailed verification analysis to demonstrate the expected benefits. Our verification study used a wire-in-a-void model to create a non-trivial test problem for which we could derive an analytical solution. We derived one using perturbation theory, but found that it was not uniformly accurate and could not satisfy our verification study, so we formulated a new analytical solution using the method of manufactured solutions, and were able to show that the new formulation performed as expected. With the new formulation verified, we briefly discussed a few additional results and some research on the exploding wire problem using the new formulation.

REFERENCES

[1] C. C. Ashcraft and A. C. Robinson, *Verification of an Improved Cylindrical Magnetic Diffusion Algorithm in ALEGRA*, tech. rep., SNL, in preparation. 2015.

[2] J. Melissen and J. Simkin, *A New Coordinate Transform for the Finite Element Solution of Axisymmetric Problems in Magnetostatics*, IEEE Transactions on Magnetics, 26 (1990), pp. 391–394.

NEW DEVELOPMENTS IN USING SCHWARZ METHODS FOR MODEL COUPLING

JAMES CHEUNG*, MAURO PEREGO†, AND PAVEL B. BOCHEV‡

Abstract. In this proceeding report, we discuss our progress of the two projects we have worked on during the 2015 Summer Intern Program at CSRI. We divide our report in two sections: local extension formulation for coupling mismatching interfaces, and the classical Schwarz method for coupling classical density functional theory and the Poisson-Nernst-Planck equation. Despite the fact that both of these projects solve very distinct problems, they share a common design principle rooted in alternating Schwarz procedures.

In the first part of the report, we present the concept of using extensions to bridge gaps in general noncoincident interface coupling problems. We show that using the Steklov-Poincaré framework and the Dirichlet-Neumann iterative method, we are able to pass a linear patch test while converging at the same rate as the Dirichlet-Neumann method for coincident interfaces.

In the second part of the report, we demonstrate the potential of the classical Schwarz iterative method to couple classical Density Functional Theory to the Poisson-Nernst-Planck equations in one dimension. The cDFT-PNP coupling project is an ongoing endeavor, and we hope that in the near future we will be able to demonstrate this coupling in two dimensions.

1. Local extension formulation for coupling mismatching interfaces.

1.1. Introduction. In many computational applications, boundary value problems (BVPs) are solved on separate domains coupled through a sharp interface. Often, separate discretization of the problem domains result in non-matching interfaces. The mismatch of the discrete interfaces leads to overlaps and voids that make formulating a coupling method a difficult endeavor. Typically, the mismatch occurs as a result of meshing requirements for specific problems. An example of this is the coupling of fluid equations with structural equations, where a finer mesh is required at the interface of the fluid domain to capture boundary layer effects, while a coarse mesh is sufficient to model the mechanics of the structure. Currently, the most common methods for dealing with mismatching interfaces involve meshing the gaps at the mismatched interface, and the projection of quantities from a "master" interface to a "slave" interface [12, 2]. While these methods allow the transfer of quantities from one mismatched interface to another, oftentimes they are not trivial to implement and may have an adverse impact on the computational efficiency. In addition, they may also induce oscillations, or "ghost forces", in the neighborhood of the interface.

In this part of the report, we present a simple method to couple BVPs with mismatching interfaces using local Taylor series extensions. We use a Steklov-Poincaré framework [11, 17] to couple the extension data between two models. While the focus of this work is to bridge gaps at the noncoincident interface using extensions, we also demonstrate that application of our approach to mesh overlaps also yields an accurate solution. We will also demonstrate that this method fulfills the desirable requirement of passing a linear patch test. We focus on piecewise linear finite elements in this work.

1.2. Notation. Let Ω_i, $i = 1, 2$ be the subdomains where the boundary value problems are posed and u_i, $i = 1, 2$ to be the solution of the said BVPs. The coupled solution $u = u_1\big|_{\Omega_1} + u_2\big|_{\Omega_2}$ lives on the domain $\Omega = \Omega_1 \cup \Omega_2$. We denote $\Gamma = \partial\Omega_1 \cap \partial\Omega_2$ to be the interface where we wish to enforce continuity of u_1 and u_2.

In our discrete problem, we denote Ω_i^h, $i = 1, 2$, as the triangulation of Ω_i and u_i^h, to be the discrete solution of the BVPs. We denote the coupled solution $u^h = u_1^h\big|_{\Omega_1^h} + u_2^h\big|_{\Omega_2^h}$,

*Department of Scientific Computing, Florida State University, jc07g@my.fsu.edu
†Sandia National Laboratories, mperego@sandia.gov
‡Sandia National Laboratories, pbboche@sandia.gov

where u^h lives on the coupled domains $\Omega^h = \Omega_1^h \cup \Omega_2^h$. We denote Γ_i^h, to be the discrete interfaces where we wish to approximate continuity of our solutions. Generally, $\Gamma_1^h \neq \Gamma_2^h$ since we are considering separate discretizations of Ω_i.

We will define operators $\mathcal{E}_i$, to be extension operators that extend the domain of u_i into $\hat{\Omega}_i$, where we have denoted $\hat{\Omega}_i$ as the extended domain such that $\partial\hat{\Omega}_1 \cap \partial\Omega_2 = \Gamma_2$, $\partial\hat{\Omega}_2 \cap \partial\Omega_1 = \Gamma_1$, and $\partial\hat{\Omega}_i \cap \partial\Omega_i = \partial\Omega_i \setminus \Gamma_i$, where $i = 1, 2$. Likewise, we will denote $\mathcal{E}_i^h$, to be extension operators that extend the domain of u_i^h into $\hat{\Omega}_i^h$, where $\hat{\Omega}_i^h$ is a discretization of $\hat{\Omega}_i$.

In our report, we use the shorthand notation $\partial_{\mathbf{n}_i} v = \nabla v \cdot \mathbf{n}_i$, to denote the normal derivative relative to the interfaces Γ_i. We also denote $a_i^h(\mu^h, \eta^h) = \left(k_i \nabla\mu^h, \nabla\eta^h\right)_{\Omega_i^h}$ as the discrete bilinear form associated with (1.1), and $(\mu, \eta)_X$ is the standard $L^2(X)$ inner product, where X is an arbitrary domain.

1.3. Mathematical background. In our report, we use the Poisson equation as our prototypical boundary value problem.

$$\begin{cases} \nabla \cdot (k_i(\mathbf{x})\nabla u_i) = f_i \text{ on } \Omega_i \\ \qquad\qquad\qquad u_i = 0 \ \text{ on } \partial\Omega_i \setminus \Gamma, \end{cases} \tag{1.1}$$

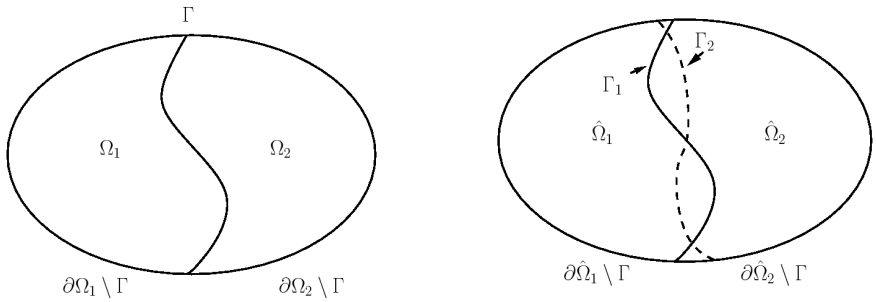
$i = 1, 2$, subject to the appropriate interface conditions given by

$$\begin{cases} u_1 - u_2 = 0 \text{ on } \Gamma \\ k_1 \partial_{\mathbf{n}_1} u_1 + k_2 \partial_{\mathbf{n}_2} u_2 = 0 \text{ on } \Gamma. \end{cases} \tag{1.2}$$

Here, k_i denotes tensor or constant material properties restricted to Ω_i, and f_i is a forcing function with the same restriction. The system (1.1) and (1.2) give us what is often called a *Steklov–Poincaré* in the domain decomposition literature [11].

1.3.1. Analytic extensions. A function is said to be *analytic* if the function can be represented by a power series locally at every point in the domain. It is well known that analytic functions can be uniquely *extended* to a larger domain, sometimes $\mathbb{R}^n$, using power series approximations on limit points of the current domain. While all functions are not analytic, certain classes of functions admit extensions. It is generally known that functions in $L^p(\Omega)$ can be locally approximated using $C_c^\infty(\Omega)$ functions [1], and because of this property, it is possible to create an unique extension into some finite extended domain. The size of this unique extension is often called the *radius of extension*, and it is inversely proportional to the smoothness of the function in some neighborhood containing the boundary of the domain. The method described in this report relies on smoothness of the solution near Γ, so that the radius of extension is large enough to couple with the adjoining noncoincident interface. An analysis of the size of the extension will be discussed in a future work, For the time being, uniqueness and existence of a sufficiently large local extension is assumed.

1.4. Rationale for using extensions. In the analytical formulation of general interface coupling problems, the problem is formulated such that the interfaces of Ω_1 and Ω_2 match exactly. However, because numerical solutions of (1.1)-(1.2) requires some type of domain discretization, such as meshing, often occurs that the discrete interfaces do not match exactly. One example of this is when Ω_1 and Ω_2 are meshed separately. This typically leaves voids and overlaps in the domain which do not exist in the analytical domains. The fact that the analytical solution exists in the void regions suggests that there may be a method to approximate the solution in the voids. Herein lies the premise of our method: to approximate the solution across the voids of the discrete interface by extension of the discrete solution.



(a) Coincident Interface (b) Noncoincident Interfaces

1.5. Coupling mismatched interfaces. For now, let us assume that we are solving BVPs exactly on $\Omega_i = \Omega_i^h$ with $\Gamma_i = \Gamma_i^h$ with $i = 1, 2$. Let us denote $\tilde{u}_i$ as the solution on these domains with noncoincident interfaces, and u_i to be the "true" solution, e.g, the solution of the matching interface coupling.In general we cannot couple the trace of $\tilde{u}_1$ and $\tilde{u}_2$ directly because $\Gamma_1^h \neq \Gamma_2^h$. The same issue exists with the natural boundary conditions $k_i \partial_{\mathbf{n}_1} \tilde{u}_i^h\big|_{\Gamma_i}$, $i = 1, 2$. To remedy this problem, we choose to use extension operators to extend the values of $\tilde{u}_2$ to Γ_1 and $k_1 \partial_{\mathbf{n}_1} \tilde{u}_1$ to Γ_2. The hybrid formulation for the noncoincident interface problem becomes

$$\begin{cases} \nabla \cdot (k_i(\mathbf{x}) \nabla \tilde{u}_i) = \mathcal{E}_i f_i \text{ on } \Omega_i \\ \qquad\qquad \tilde{u}_i = 0 \qquad \text{on } \displaystyle\bigcup_{i=1,2} \partial\Omega_i \setminus \Gamma_i, \end{cases} \tag{1.3}$$

$i = 1, 2$, subject to

$$\begin{cases} \tilde{u}_1 - \mathcal{E}_2 \tilde{u} = 0 \text{ on } \Gamma_1 \\ \mathcal{E}_1 k_1 \partial_{\mathbf{n}_1} \mathcal{E}_1 \tilde{u}_1 + k_2 \partial_{\mathbf{n}_2} \tilde{u}_2 = 0 \text{ on } \Gamma_2. \end{cases} \tag{1.4}$$

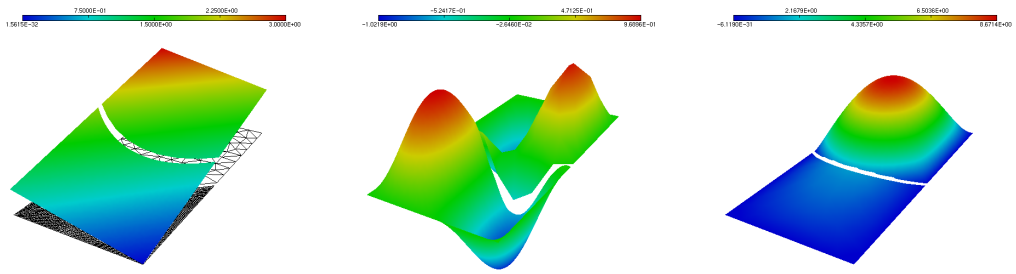
If we are solving the same boundary value problem on both Ω_1 and Ω_2, we see analytically that the above hybrid formulation will give us the exact solution since the overlapping extensions are equivalent. Consequently, this formulation provides an appropriate foundation for a mesh repair method.

However, if we choose to solve different BVPs on Ω_1 and Ω_2, there will be an error associated with the interface conditions (1.4), since we are enforcing continuity of $\tilde{u}_1$ and $\tilde{u}_2$ on Γ_1 and continuity of the natural boundary conditions on Γ_2. Because Γ_1 and Γ_2 are separated by a gap, we assume that $||u_i - \tilde{u}_i||_{0,\Gamma_i} \leq M_{i,1}\varepsilon$, where $\varepsilon = \sup\limits_{\mathbf{x}\in\Gamma_{1,2}} \min\limits_{\mathbf{x}'\in\Gamma_{2,1}} |\mathbf{x} - \mathbf{x}'|$ is the maximum distance of the nearest points on Γ_1 and Γ_2, $M_{i,1}$ is a positive constant. Similarly, we expect $||k_i \partial_{\mathbf{n}_i} u_i - k_i \partial_{\mathbf{n}_i} \tilde{u}_i||_{0,\Gamma_i} \leq M_{i,2}\varepsilon$.

Since it is most common to use piecewise linear functions to approximate Γ_i^h in finite element approximations, we will assume that $\epsilon < Kh^2$ [16], where h is the maximum diameter of the approximating elements of Γ_i^h. Because the solution of elliptic PDEs are bounded above by its data, we expect

$$||u_i - \tilde{u}_i^h||_{0,\Omega_i^h} \leq C_i h^2 \tag{1.5}$$

Fig. 1.1: Matching various problems between fine and coarse meshes across gaps between discrete parabolic interfaces using Taylor series extensions



(a) Linear patch test

(b) Solution of mesh repair problem

(c) Solution of transfer problem

for our approximation. Since piecewise linear finite element methods are second order accurate in the $L^2(\Omega_i)$ norm, we see that the error induced by this formulation is within the discretization error of the linear piecewise finite element method. Our numerical experiments confirm the error behavior supplied by this heuristic argument. A more rigorous analysis will be provided in a later publication.

1.5.1. Constructing a local extension. It is well known that the weak solution of the second order elliptic partial differential equation has $H^1(\Omega)$ regularity, with possiblility of higher regularity [6]. Because $H^1(\Omega_i)$ is the completion of $C_c^\infty(\Omega_i)$ functions with respect to the $H^1(\Omega_i)$ norm, the construction of a local extension using Taylor series is defined. For every point $\mathbf{x} \in \Gamma_1$, the Taylor series extension is defined as.

$$\mathcal{E}_2 u_2(\mathbf{x}) = u_2(\mathbf{x}') + \nabla u_2(\mathbf{x}') \cdot (\mathbf{x} - \mathbf{x}') + (\mathbf{x} - \mathbf{x}')^T \nabla\nabla u_2(\mathbf{x}')(\mathbf{x} - \mathbf{x}') + \cdots \tag{1.6}$$

where the points $\mathbf{x}'$ are chosen such that $\mathbf{x}' = \underset{\mathbf{x}' \in \Gamma_2}{\arg\min} |\mathbf{x} - \mathbf{x}'|$. Similarly, for every point $\mathbf{x}' \in \Gamma_2$

$$\mathcal{E}_1 u_1(\mathbf{x}') = u_1(\mathbf{x}) + \nabla u_1(\mathbf{x}) \cdot (\mathbf{x} - \mathbf{x}') + (\mathbf{x} - \mathbf{x}')^T \nabla\nabla u_1(\mathbf{x})(\mathbf{x} - \mathbf{x}') + \cdots \tag{1.7}$$

where $\mathbf{x} = \underset{\mathbf{x} \in \Gamma_1}{\arg\min} |\mathbf{x} - \mathbf{x}'|$ for every point $\mathbf{x}'$.

We define the linear finite element extensions by

$$\begin{aligned} \mathcal{E}_1^h u_1^h(\mathbf{x}_q') &= u_1^h(\mathbf{x}_p) + \nabla u_1^h(\mathbf{x}_p) \cdot (\mathbf{x}_q' - \mathbf{x}_p) \\ \mathcal{E}_2^h u_2^h(\mathbf{x}_p) &= u_2^h(\mathbf{x}_q') + \nabla u_2^h(\mathbf{x}_q') \cdot (\mathbf{x}_p - \mathbf{x}_q') \end{aligned} \tag{1.8}$$

where the vertices $\mathbf{x}_p \in \Gamma_1^h$ are the nearest neighbors of the vertices $\mathbf{x}_q' \in \Gamma_2^h$. Assuming that $\Omega_i = \Omega_i^h$, we see that

$$\begin{aligned} \mathcal{E}_1 u_1(\mathbf{x}_q') - \mathcal{E}_1^h u_1^h(\mathbf{x}_q') &= [u_1(\mathbf{x}_p) - u_1^h(\mathbf{x}_p)] + [\nabla u_1(\mathbf{x}_p) - \nabla u_1^h(\mathbf{x}_p)] \cdot (\mathbf{x}_q' - \mathbf{x}_p) \\ &\quad + Q_2 |\mathbf{x}_q' - \mathbf{x}_p|^2 \\ \mathcal{E}_2 u_2(\mathbf{x}_p) - \mathcal{E}_2^h u_2^h(\mathbf{x}_p) &= [u_2(\mathbf{x}_q') - u_2^h(\mathbf{x}_q')] + [\nabla u_2(\mathbf{x}_q') - \nabla u_2^h(\mathbf{x}_q')] \cdot (\mathbf{x}_p - \mathbf{x}_q') \\ &\quad + Q_1 |\mathbf{x}_p - \mathbf{x}_q'|^2. \end{aligned} \tag{1.9}$$

By taking the $L^2(\Gamma_i)$ norm, and using finite element error bounds, we get

$$\begin{aligned} ||\mathcal{E}_1 u_1 - \mathcal{E}_1^h u_1^h||_{0,\Gamma_2} &\leq A_1 h_1^2 + \sup|\mathbf{x}_q' - \mathbf{x}_p| B_1 h + C_1 |\mathbf{x}_q' - \mathbf{x}_p|^2 \\ ||\mathcal{E}_2 u_2 - \mathcal{E}_2^h u_2^h||_{0,\Gamma_1} &\leq A_2 h_2^2 + \sup|\mathbf{x}_p - \mathbf{x}_q'| B_2 h + C_2 |\mathbf{x}_p - \mathbf{x}_q'|^2. \end{aligned} \tag{1.10}$$

Finally, since we assume that $\sup|\mathbf{x}_q' - \mathbf{x}_p| = \epsilon \leq Mh^2$, we arrive at our final result

$$\begin{aligned} ||\mathcal{E}_1 u_1 - \mathcal{E}_1^h u_1^h||_{0,\Gamma_2} &\leq K_1 h_1^2 \\ ||\mathcal{E}_2 u_2 - \mathcal{E}_2^h u_2^h||_{0,\Gamma_1} &\leq K_2 h_2^2. \end{aligned} \tag{1.11}$$

The above inequality suggests that the error of the extension has the same order accuracy as the linear piecewise approximation.

1.5.2. Coupling algorithm. To solve the hybrid formulation (1.3), we use a Dirichlet-Neumann iterative method [11, 17] to transfer data between the two domain interfaces.

ALGORITHM 1. *Dirichlet-Neumann with Extensions*

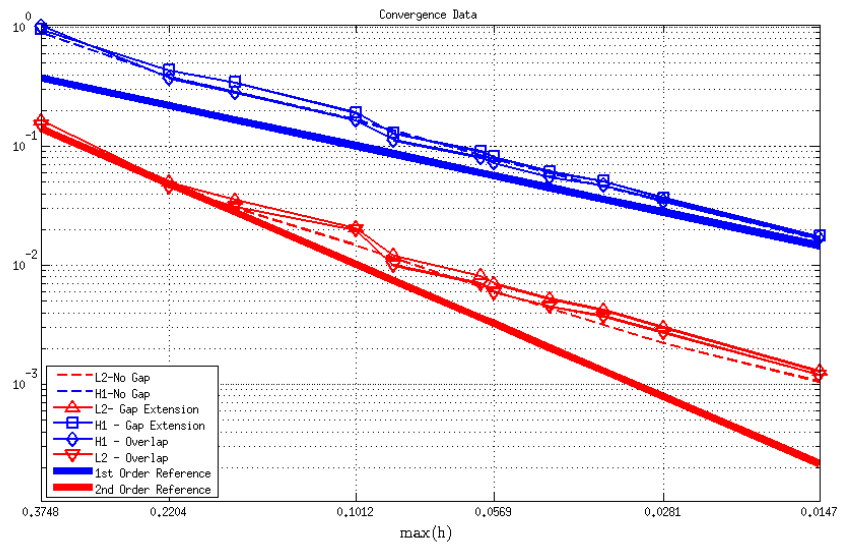
1. *Denote* $\{\mathbf{x}_p\}_{p=1}^P$ *as the set of nodes on* Γ_1^h *and* $\{\mathbf{x}_q\}_q^Q$ *as the set of nodes on* Γ_2^h.
2. *For every* $\mathbf{x}_p \in \Gamma_1^h$ *find the nearest node* $\mathbf{x}_q = \arg\min_{\mathbf{x}_q} |\mathbf{x}_p - \mathbf{x}_q|$.
3. *Begin with an initial Dirichlet guess* $\theta_1^{(0)}$.
4. *For* $k = 0, 1, 2, \ldots$
   (a) *Solve* $a_1^h(u_1^{h(k)}, v_1^h) = (f_1^h, v_1^h)_{\Omega_1^h}$ *with* $u_1^{h(k)}\big|_{\Gamma_1^h} = \theta_1^{(k)}$.
   (b) *Construct* $\mathcal{E}_1^h u_1^{h(k)}\big|_{\Gamma_2^h}$.
   (c) *Set* $g_2^{(k)} = \partial_{\mathbf{n}_1} \mathcal{E}_1 k_1 u_1^{h(k)}\big|_{\Gamma_2^h}$.
   (d) *Construct* $\mathcal{E}_2^h u_2^{h(k)}\big|_{\Gamma_1^h}$.
   (e) *Solve* $a_2^h(u_2^{h(k)}, v_2^h) = (f_2^h, v_2^h)_{\Omega_2^h} - (g_2^{(k)}, v_2^h)_{\Gamma_2^h}$.
   (f) *Set* $\theta_1^{(k+1)} = \mathcal{E}_2 u_2^{h(k)}\big|_{\Gamma_1^h}$.

This algorithm is simple to implement since it requires only the additional step of constructing the local extension on top of the already existing nearest neighbor search algorithm. The test function derivatives required to compute the local extension is readily available in many finite element codes.

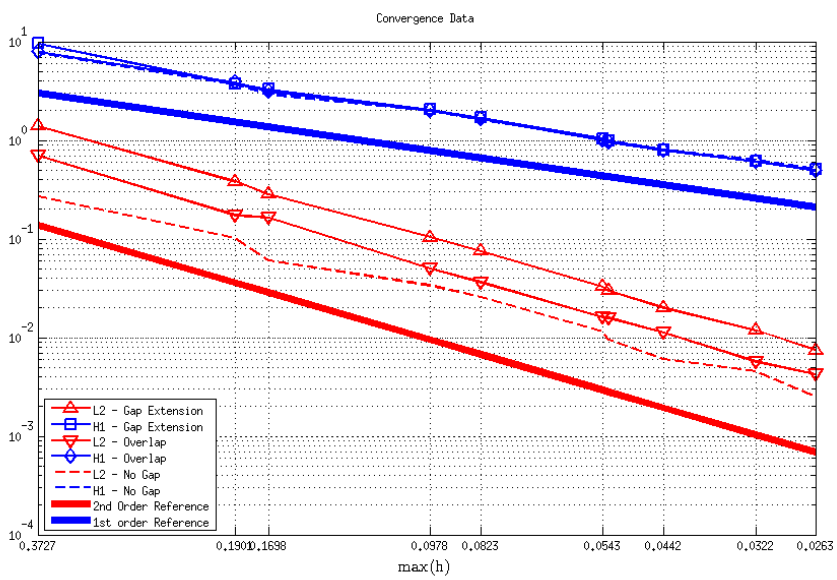
1.6. Numerical results. This section will focus on the numerical results of the method described in Section 1.5. The results will be presented in three parts: successful recovery of a linear solution, convergence of mesh tying solution, and the convergence of the solution of a transfer problem with a noncoincident interface. For all numerical results, we present results for coincident interfaces, overlapping interfaces, and gap interfaces. Demonstrating convergence for complete overlaps and complete voids is sufficient to show that this method will succeed in coupling general noncoincident interface problems. The size of the gap and overlap is taken to be h_{max}^2, where h_{max} is the diameter of the largest element in $\{\Omega_1^h, \Omega_2^h\}$. This is to mimic the error of discretizing the exact interface. It should be noted that in the overlap case, the "extension" is just an approximation of the solution at an interior point of u_1^h on Γ_2^h and u_2^h on Γ_1^h. We demonstrate that if Algorithm 1 is applied to the overlap case, we are still able to recover our coupled solution. This observation also demonstrates the low implementation complexity of our method since we need not worry if a given node at the discrete interface is an overlapping node or a void node.

1.6.1. Linear patch test. Successful recovery of a linear solution is a minimum requirement for a coupling scheme using piecewise linear elements. Many existing methods

Fig. 1.2: Convergence rates of the mesh repair and noncoincident transfer problem. The blue lines represent $||u - u^h||_{1,\Omega}$ data and the red lines represent $||u - u^h||_{0,\Omega}$ data. The thick lines represent the first and second order reference lines.

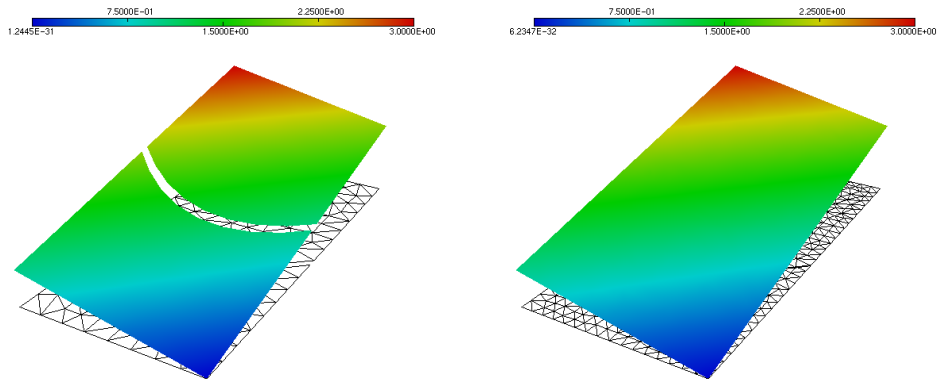


(a) Mesh Repair



(b) Noncoincident Transfer Problem

Fig. 1.3: Illustration of linear patch test results.



(a) Linear patch test with gap (b) Linear patch test with overlap

used in engineering applications do not pass this so-called "patch test". However, because we extend our solution using Taylor series, it is apparent that we are able to recover any linear function since we use the gradient information in our extension of u_i^h.

1.6.2. Mesh repair problem. In this test problem, we consider the manufactured solution $u(x, y) = \sin(\pi x)\sin(\pi y)$, given as the solution of

$$\begin{cases} -\Delta u = -\pi^2 \sin(\pi x)\sin(2\pi y) - 4\pi^2 \sin(\pi x)\sin(2\pi y) & \text{in } \Omega \\ \quad u = 0 & \text{on } \partial\Omega. \end{cases} \tag{1.12}$$

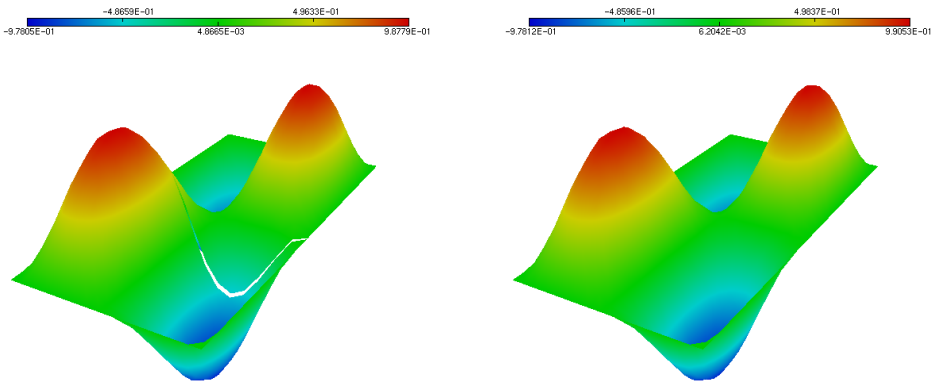
We decompose and then discretize the domain Ω into Ω_1^h and Ω_2^h, where both subdomains are "joined" by noncoincident interfaces, Γ_1^h and Γ_2^h. Here, we choose $\Omega_1^h \cup \Omega_2^h = [-1, 1] \times [-0.5, 0.5]$. A parabolic interface given by the parametrization $(x = -t^2, y = t)$, $t = [-0.5, 0.5]$, was used in this test problem. It is observed that our method has first order convergence in the $H^1(\Omega)$ and $L^2(\Omega)$ norm. However, while second order convergence in $L^2(\Omega)$ is optimal, we see the same behavior for coincident interfaces. This implies that the sub-optimal convergence is caused not by our extension, but the Dirichlet-Neumann iterative method itself. It is also observed that the slope and the additive constants of the log-plot for the error plot are similar for the gap, overlap, and coincident test problems. This suggests that the error of the extension overlap is negligible compared to the overall error of the approximation.

1.6.3. Transfer problem with noncoincident interfaces. For this test, we couple

$$\begin{cases} -\Delta u_1 = 100 & \text{in } \Omega_1 \\ \quad u_1 = 0 & \text{on } \partial\Omega_1 \setminus \Gamma \\ \quad u_1 = u_2 & \text{on } \Gamma \end{cases} \quad \begin{cases} -10\Delta u_2 = 100 & \text{in } \Omega_2 \\ \quad u_2 = 0 & \text{on } \partial\Omega_2 \setminus \Gamma \\ 10\partial_{\mathbf{n}_2} u_2 = -\partial_{\mathbf{n}_1} u_1 & \text{on } \Gamma. \end{cases} \tag{1.13}$$

Of course, because we discretize Ω_1 and Ω_2 separately, our subomdains are again "joined" by noncoincident interfaces Γ_1^h and Γ_2^h. Here, the problem domain is $\Omega_1^h \cup \Omega_2^h = [-1, 1] \times$

Fig. 1.4: Illustration of solutions of mesh repair problem



(a) Solution with gap

(b) Solution with overlap

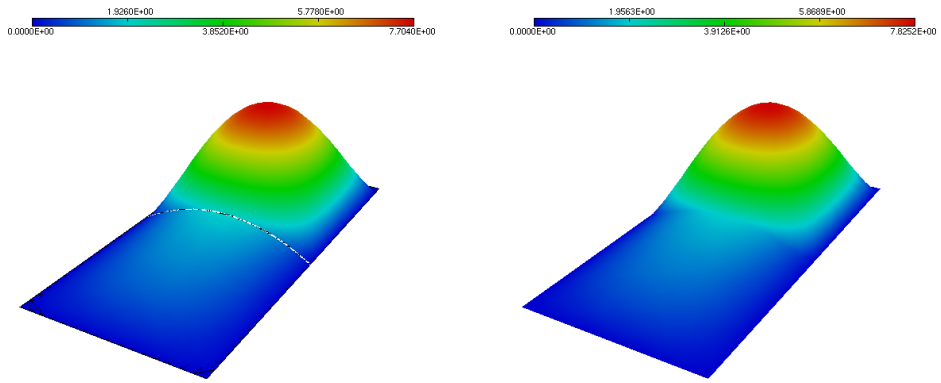
$[0, 1]$. Though it is possible to couple curved interfaces, as illustrated in Figure 1.2(c), a straight interface is used for this test problem to avoid difficulties with differing mesh discretizations of curved interfaces when comparing the coarse approximation to a fine-mesh solution. The approximation is compared to a fine-mesh solution computed using $h_{max} = 1/20$ piecewise quartic elements. We observe optimal finite element convergence, that is, first order convergence in $H^1(\Omega)$ and second order in $L^2(\Omega)$. While the convergence rates may be optimal, it is shown in Figure 1.3(b) that the additive constants of the convergence lines are different. This shows that both gap extensions and overlap errors have an additional multiplicative error. This implies that there is some sort of overlap error that contributes to the total error of the scheme. This behavior was predicted in Section 1.5.

1.7. Concluding remarks. It was demonstrated in this work that extensions can be used to solve general noncoincident interface coupling problems by using a Dirichlet-Neumann method. While this method does have an extra error of overlapping the extensions, this error does not hinder the convergence of the coupled solution for general mismatched $\mathcal{O}(h^2)$ interface approximations. We have also shown that this method passes the linear patch test, and thus spurious oscillations are less likely to occur when using this method. Furthermore, we have shown that this coupling method essentially has the same convergence properties as the standard Dirichlet-Neumann iterating method with coincident interfaces.

While there are many advantages to using our method, there are also some limitations. It was observed that the method becomes unstable if the size of the gap exceeds a certain length, τ, where $\tau >> h^2$; however, for all cases discussed in this report, the method was completely stable. In practice, this may not be a big issue since the size of the gap is often $\mathcal{O}(h_{max}^2)$. The second limitation is the coupling of solutions that have non-smooth behavior near the interface. In this case, there may not be a valid discrete extension, and we are left with the noncoincident interface to couple. These problems will be addressed in future works.

In conclusion, our work suggests that application of local extensions bears significant promise in general noncoincident coupling problems since it is easy to implement and it

Fig. 1.5: Illustration of solution of transfer problem



(a) Solution with gap (b) Solution with overlap

removes many difficulties that arise from existing coupling methods while also retaining the accuracy of the Dirichlet-Neumann iterative method for coincident interface coupling problems.

1.8. Future directions. The work presented in this report demonstrates the ability to use extensions to couple noncoincident interfaces. While the Poincaré-Steklov framework is shown to be effective in practical cases, it really only demonstrates the preliminary effectiveness of using extensions in the noncoincident interface coupling setting. In the future, we will explore using an optimal control based coupling framework to minimize the functional

$$\min_{u_1,u_2} \mathcal{J}[u_1,u_2] = \frac{1}{2}\int_\Gamma \left(\mathcal{E}_1 u_1 - \mathcal{E}_2 u_2\right)^2 d\Gamma + \frac{\beta}{2}\int_\Gamma \left(\partial_{\mathbf{n}_1}\mathcal{E}_1 u_1 + \partial_{\mathbf{n}_2}\mathcal{E}_2 u_2\right)^2 d\Gamma \tag{1.14}$$

subject to

$$\begin{cases} -k_1\Delta u_1 = f_1 & \text{in } \Omega_1 \\ k_1\partial_{\mathbf{n}_1} u_1 = g_1 & \text{on } \gamma_1 \\ u_1 = 0 & \text{on } \partial\Omega_1 \setminus \gamma_1 \end{cases} \qquad \begin{cases} -k_2\Delta u_2 = f_2 & \text{in } \Omega_2 \\ k_2\partial_{\mathbf{n}_2} u_2 = g_2 & \text{on } \gamma_2 \\ u_2 = 0 & \text{on } \partial\Omega_2 \setminus \gamma_2, \end{cases} \tag{1.15}$$

where β is a penalization parameter, γ_i denotes the "virtual" interface defined as $\partial\Omega_i \cap \gamma_i$, and Γ denotes the "true" interface where we wish to enforce continuity of the coupled solution. The flexibility of this formulation allows us to choose the true interface to be one of the virtual interfaces, which will be useful if one of the solutions is not effectively extensible. This formulation eliminates extension overlaps and makes convergence analysis simpler by posing the coupling problem in a well-studied optimization framework. We will also analyze the stability and accuracy of the extensions as well as demonstrate the ability of the extension to reproduce any p-order polynomial with $(p+1)$-order finite element methods. The uniqueness of the extension must also be proven in order to justify rigorously the use of extensions in the coupling framework. Finally, we will apply this formulation to fluid-structure interactions to demonstrate its effectiveness in preserving forces and matching displacements.

2. The Classical Schwarz Method for Coupling Classical Density Functional Theory and the Poisson Nernst Planck Equation.

2.1. Introduction. The premise of Classical Density Functional Theory (cDFT) relies on the observation that the behavior of hard sphere particles can be modeled by minimizing a total energy functional with respect to the number densities in the system. While the formulation of cDFT may accurately describe many features of fluid particle particle behavior, computing its solution is not trivial. The complexity of the first variation with respect to number densities is proportional to the number of physics terms included in the formulation. The resulting model is a highly nonlinear system of integral and differential equations. Also while, cDFT is very useful for capturing oscillations and nonlocal effects of the number density in a small region near a wall, its usefulness diminishes outside this region. For this reason, we are interested in developing a method that couples this nonlocal model with a local model away from a wall to reduce the cost of determining the microscale behavior of a fluid in a larger domain. In this report, we present a preliminary result from coupling nonequilibrium steady-state cDFT with the simpler Poisson-Nernst-Planck equations using the classical alternating Schwarz method in one dimension.

2.2. Mathematical background. The steady-state Poisson-Nernst-Planck (PNP) equation is given by

$$\begin{cases} \nabla \cdot (D_\alpha \rho_\alpha \nabla \beta \mu_\alpha) \text{ on } \Omega_p \\ -\nabla \cdot (\epsilon(\mathbf{x}) \nabla \phi) = \dfrac{1}{\epsilon_0} \displaystyle\sum_\alpha^N e z_\alpha \rho_\alpha \text{ on } \Omega_p \end{cases} \tag{2.1}$$

where Ω_p denotes the PNP domain, α is the particle index, ρ_α is the density, ϕ is the electrostatic potential, D_α is the diffusion constant, e is the electron charge, and z_α is the valency of the particle. We have defined $\beta\mu_\alpha = \log(\rho_\alpha) + z_\alpha \phi$ as a placeholder variable to represent the chemical potential. This model describes the drift-diffusion behavior of point charge particles in a fluid through their electrostatic interactions. Dirichlet boundary conditions are usually presecribed for ρ_α and ϕ for (2.1).

For brevity, we omit a detailed review of cDFT and refer the reader to [15]. In a broad sense, the steady-state nonequilibirum cDFT formulation generalizes the PNP model into the nonlocal setting by adding nonlocal charged hard-sphere properties of particles in the fluid. The principle distinction between the PNP formulation and the cDFT formulation is that the chemical potential is defined as

$$\beta\mu_\alpha = \log(\rho_\alpha) + \beta V + \frac{\delta \beta F^{ex}}{\delta \rho_\alpha} + z_\alpha \phi, \tag{2.2}$$

where F^{ex} are the excess free energy contributions of the system determined by a grand canonical ensemble functional, V is the external potential, and $\beta = (k_b T)^{-1}$, where k_b is the Boltzmann constant, and T is the absolute temperature. Essentially, the steady-state nonequilibrium formulation of cDFT is (2.1) using (2.2) coupled with an integral formulation of $\frac{\delta \beta F^{ex}}{\delta \rho_\alpha}$. In most applications, Dirichlet boundary conditions are prescribed for μ_α and ϕ_α. Because of the nonlocal equation of the density equation, a volume constraint must be enforced for ρ_α in the interaction region.

2.3. Schwarz Method. Numerical simulations suggest that solutions of cDFT and PNP models are almost indistinguishable in regions that do not contain nonlocal behavior in the cDFT solution. This motivates our approach to use the Schwarz method as a coupling mechanism between cDFT and PNP through an overlap region. For the remainder of the

report, the problem domains will be denoted Ω_p and Ω_d for the PNP and cDFT domains respectively. Likewise Γ_p and Γ_d will be the part of the boundaries that receive Dirichlet information from the adjacent overlapping domain. In addition, we will denote $\tilde{\Omega}_d$ to be the part of the interaction region of the cDFT domain that receives volume constraint information from the PNP solution. Finally, we denote $\Omega_o = \Omega_p \cap \Omega_d$ to be the overlapping region. The algorithm used to couple the PNP and DFT solutions is presented below. Tramonto was used to carry out the computations for obtaining numerical solutions to the cDFT and PNP equations.

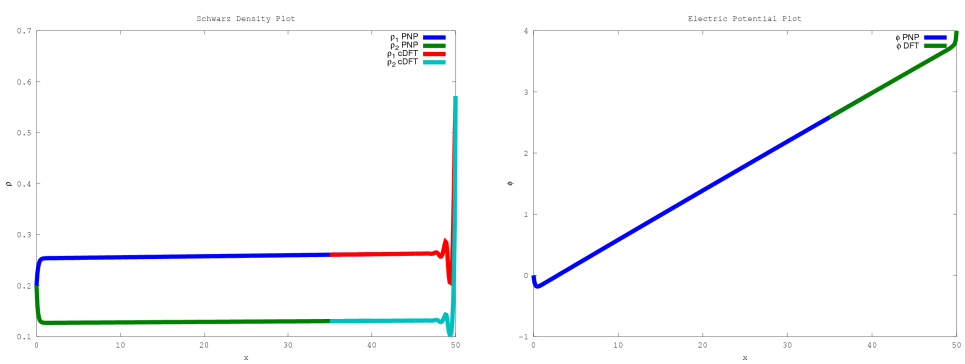
ALGORITHM 2. *cDFT-PNP Coupling using Schwarz*

1. *Set Dirichlet boundaries* $\{\rho_\alpha^{p,(0)}, \phi^{p,(0)}\}\big|_{\Gamma_p} = \{\theta_p^{\rho_\alpha}, \theta_p^{\phi}\}$ *as initial guesses.*
2. *Solve PNP equation for* $\{\rho_\alpha^{p,(0)}, \phi^{p,(0)}\}$
3. *Until convergence:*
   (a) *Set* $\rho_\alpha^{d,(k+1/2)}\big|_{\tilde{\Omega}_p} = \rho_\alpha^{d,(k)}\big|_{\tilde{\Omega}_d \cap \Omega_p}$.
   (b) *Set* $\phi^{d,(k+1/2)}\big|_{\Gamma_d} = \phi^{p,(k)}\big|_{\Gamma_d \cap \Omega_p}$.
   (c) *Solve cDFT equation for* $\{\rho_\alpha^{d,(k+1/2)}, \phi^{d,(k+1/2)}, \mu_\alpha^{d,(k+1/2)}\}$
   (d) *Set* $\{\rho_\alpha^{p,(k+1)}, \phi^{p,(k+1)}\}\big|_{\Gamma_p} = \{\rho_\alpha^{d,(k+1/2)}, \phi^{d,(k+1/2)}\}\big|_{\Gamma_d}$
   (e) *Solve PNP equation for* $\{\rho_\alpha^{p,k+1}\, \phi^{p,(k+1)}\}$.

In the above algorithm, $(\cdot)^d$ denotes a cDFT variable, and $(\cdot)^p$ denotes a PNP variable.

2.4. Preliminary numerical results and observations. In the test problem, we solve the PNP equation in $x \in [0, 40]$ and the DFT equation in $x \in [35, 50]$ with the overlap region being $x \in [35, 40]$. We implement "wall" boundary conditions on the right boundary to induce nonlocal oscillations near the right domain. In mathematical terms, the wall conditions means that we have set $\rho_\alpha^d = 0$ in the interaction region in $x > 50$ and $\mu_\alpha^d(50) = -\infty$. This also means that the particle exists only at radius-length away from the wall. We also set $\phi^d(50) = 4$. On the right domain, $\rho_\alpha^p(0) = 0.2$ and $\phi^p(0) = 0$. In this test problem, we have placed a $+1$ charge on the $\alpha = 1$ species, and a -2 charge on the $\alpha = 2$ species. While this test problem may not have any physical significance, it highlights, in the abstract sense, the ability to perform a cDFT-PNP coupling using the classical Schwarz method.

Fig. 2.1: The coupled solution after 10 iterations.



(a) cDFT-PNP density (b) cDFT-PNP electrostatic potential

It was found that for this test problem, and for other 1D test problems that we can couple

the density and electric potentials; however, the chemical potentials remain uncoupled. This is expected due to the general non-uniqueness of potentials and due a current lack of a coupling mechanism to match chemical potentials from the PNP solution to the cDFT solution. This is partly due to the fact that in PNP, the chemical potential is not treated as a variable, while the opposite is true in cDFT.

2.5. Concluding remarks. In this report, we have demonstrated the possibility of coupling cDFT and PNP using the classical Schwarz method. While we have only presented the a preliminary result in 1D, we believe that this method will still work in 2D and 3D due to the visibly indistinguishable behavior of cDFT and PNP solutions in regions that do not exhibit nonlocal effects in the cDFT solution. This work is an ongoing endeavor, and we hope that in the near future, a 2D cDFT-PNP coupling method will be implemented using Tramonto.

3. Acknowledgments. The authors would like to thank Max Gunzburger for his guidance during the spring semester that lead to the ideas discussed in the first part of this report. We would also like to acknowledge Paul Kuberry for his helpful instruction on using FreeFem++'s more technical features and for helpful discussions on the subject of interface coupling. In addition, we would like to thank Amalie Frischknecht for guiding us on classical Density Functional Theory and the Tramonto software. Finally, we would like to thank Michael Parks for reviewing this report.

Part of this research was carried under the auspices of the Collaboratory on Mathematics for Mesoscopic Modeling of Materials (CM4), funded by the DOE Office of Science Advanced Scientific Computing Research (ASCR) Applied Mathematics program.

REFERENCES

[1] D. Day and P. Bochev, *Analysis and computation of a least-squares method for consistent mesh tying*, Journal of Computational and Applied Mathematics, 218 (2008), pp. 21–33.

[2] A. De Boer, A. Van Zuijlen, and H. Bijl, *Review of coupling methods for non-matching meshes*, Computer methods in applied mechanics and engineering, 196 (2007), pp. 1515–1525.

[3] A. de Boer, A. H. van Zuijlen, and H. Bijl, *Comparison of conservative and consistent approaches for the coupling of non-matching meshes*, Computer Methods in Applied Mechanics and Engineering, 197 (2008), pp. 4284–4297.

[4] Q. Du and M. D. Gunzburger, *A gradient method approach to optimization-based multidisciplinary simulations and nonoverlapping domain decomposition algorithms*, SIAM journal on numerical analysis, 37 (2000), pp. 1513–1541.

[5] M. DElia and P. B. Bochev, *Optimization–based coupling of nonlocal and local diffusion models*, in MRS Proceedings, vol. 1753, Cambridge Univ Press, 2015.

[6] L. Evans, *Partial differential equations*, American Mathematical Society, Providence, RI,, 19 (2010).

[7] L. J. D. Frink and A. G. Salinger, *Two-and three-dimensional nonlocal density functional theory for inhomogeneous fluids: I. algorithms and parallelization*, Journal of Computational Physics, 159 (2000), pp. 407–424.

[8] M. Gunzburger, J. Peterson, and H. Kwon, *An optimization based domain decomposition method for partial differential equations*, Computers & Mathematics with Applications, 37 (1999), pp. 77–93.

[9] F. Hecht, *New development in FreeFem++*, Journal of Numerical Mathematics, 20 (2012), pp. 251–266.

[10] E. H. Lieb and M. Loss, *Analysis, volume 14 of graduate studies in mathematics*, American Mathematical Society, Providence, RI,, 4 (2001).

[11] T. Mathew, *Domain decomposition methods for the numerical solution of partial differential equations*, vol. 61, Springer Science & Business Media, 2008.

[12] K. Park, C. Felippa, and G. Rebel, *A simple algorithm for localized construction of non-matching structural interfaces*, International Journal for Numerical Methods in Engineering, 53 (2002), pp. 2117–2142.

[13] M. Parks, L. Romero, and P. Bochev, *A novel Lagrange-multiplier based method for consistent mesh tying*, Computer methods in applied mechanics and engineering, 196 (2007), pp. 3335–3347.

[14] M. L. Parks, P. B. Bochev, and R. B. Lehoucq, *Connecting atomistic-to-continuum coupling and domain decomposition*, Multiscale Modeling & Simulation, 7 (2008), pp. 362–380.

[15] R. Roth, *Fundamental measure theory for hard-sphere mixtures: a review*, Journal of Physics: Condensed Matter, 22 (2010), p. 063102.

[16] G. Strang and G. J. Fix, *An analysis of the finite element method*, vol. 212, Prentice-Hall Englewood Cliffs, NJ, 1973.

[17] A. Toselli and O. Widlund, *Domain decomposition methods: algorithms and theory*, vol. 3, Springer, 2005.

DEVELOPMENT OF HIGHER ORDER STRONG STABILITY PRESERVING IMPLICIT-EXPLICIT RUNGE KUTTA METHOD

SIDAFA CONDE* AND JOHN N. SHADID†

Abstract. Strong Stability Preserving (SSP) numerical time integrators for initial value ODEs preserves the monotonicity properties of the Forward Euler method in any of the norm, semi-norm, or convex functionals. In this paper, the construction and analysis of Strong Stability Preserving Implicit-Explicit (IMEX) Runge–Kutta (RK) methods for the time integration of an additive initial value problem is discussed. The goal of this study is to seek IMEX methods that have additional SSP stability properties for the implicit integrator as well as the overall IMEX method. Most previous high-order methods only have the explicit part be SSP, while enforcing the implicit scheme to be L-Stable. The new methods developed ensure the explicit and implicit methods are SSP, while enforcing the overall scheme to be SSP, additionally we constrain the implicit part to be L-stable. The new methods were constructed in search of sub-optimal SSP methods with the desired properties for the implicit and overall IMEX integrator. Though there were no overall SSP fourth order L-Stable IMEX methods, methods of order $p \leq 4$ are investigated. The accuracy and stability of these results is demonstrated on a convection-diffusion system and on a coupled first-order wave equation system.

1. Introduction. This work is concerned with the numerical solution of the initial value problem

$$y'(t) = f(y(t)) + g(y(t)), \qquad y(t_0) = y_0 \qquad t \geq t_0 \tag{1.1}$$

Here f is slow and g is fast with respect to the desired time scale of interest. Furthermore, it is assumed that $\|\cdot\|$ is a convex functional, such that for any t_0 and any solution $y(t)$ to (1.1), the following monotonicity property is satisfied

$$\|y(t)\| \leq \|y(t_0)\|, \qquad \forall t \geq t_0 \tag{1.2}$$

It is assumed that the semi-discrete form of (1.1) after spatial discretization is represented as

$$\mathbf{y}'(t) = F(\mathbf{y}) + G(\mathbf{y}), \qquad y(0) = \mathbf{y}_0 \tag{1.3}$$

Where $\mathbf{y}_0 \in \mathbb{R}^m$, and $F : \mathbb{R}^m \to \mathbb{R}^m, G : \mathbb{R}^m \to \mathbb{R}^m$ such that (1.1) has a unique solution $\mathbf{y} : [t_0, \infty) \to \mathbb{R}^m$. System (1.3) is then solved using a numerical ODE solver. Numerical methods for (1.3) compute a sequence of solutions $\mathbf{y}^1, \mathbf{y}^2, \cdots$ approximating the true solution at times $(t_0 + \Delta t), (t_0 + 2\Delta t), \cdots$. This study considers the development of a class of high order additive Implicit-Explicit Runge–Kutta methods for which the numerical solution satisfies the monotonicity property, the discrete form of (1.2) is

$$\|\mathbf{y}^{n+1}\| \leq \|\mathbf{y}^n\| \tag{1.4}$$

A common approach for such methods is to assume that the initial value problem (1.3) is monotone under Forward Euler (FE) integration, subject to some step size restrictions:

$$\|\mathbf{y} + \tau_0 F(\mathbf{y})\| \leq \|\mathbf{y}\|, \qquad \forall \mathbf{y} \in \mathbb{R}^m, \text{ for } 0 \leq \tau_0 \leq \Delta T_{FE} \tag{1.5}$$

$$\|\mathbf{y} + \tilde{\tau_0} G(\mathbf{y})\| \leq \|\mathbf{y}\|, \qquad \forall \mathbf{y} \in \mathbb{R}^m, \text{ for } 0 \leq \tilde{\tau_0} \leq \Delta \tilde{T}_{FE} \tag{1.6}$$

The term Strong Stability Preserving (SSP) is used to denote any method that gives

*University of Massachusetts Dartmouth, sconde@umassd.edu
†Sandia National Laboratories, jnshadi@sandia.gov

a solution satisfying the monotonicity condition (1.4) whenever applied to a initial value problem (1.3) which satisfies the Forward Euler conditions (1.5)-(1.6).This can be shown to hold under a step size restriction of the form

$$\Delta t \leq \min\{\mathcal{C}\Delta T_{FE}, \tilde{\mathcal{C}}\Delta \tilde{T}_{FE}\} \tag{1.7}$$

where the factor $\mathcal{C}, \tilde{\mathcal{C}}$, referred to as the SSP coefficient corresponding to F, G terms respectively, depends only on the numerical method.

An s-stage Additive Runge–Kutta (ARK) method is defined by two $s \times s$ real matrices $A, \tilde{A}$ and two real vectors $b, \tilde{b}$ such that:

$$\begin{aligned} Y_i &= \mathbf{y}^n + h\sum_{j=1}^{s} a_{ij}F_j + h\sum_{j=1}^{s} \tilde{a}_{ij}G_j, \qquad i = 1, \ldots, s \\ \mathbf{y}^{n+1} &= \mathbf{y}^n + h\sum_{i=1}^{s} b_iF_i + h\sum_{i=1}^{s} \tilde{b}_iG_i, \qquad i = 1, \ldots, s \end{aligned} \tag{1.8}$$

where Y_i are the intermediate values of the solution y at the time $t_n + c_ih$ and $F_i = F(Y_i), G_i = G(Y_i), c_i = \sum_{j=1}^{s} a_{ij}, \tilde{c}_i = \sum_{j=1}^{s} \tilde{a}_{ij}$.

The Runge–Kutta (RK) method (A, b) represents an explicit method used for the nonstiff part F, and RK method $(\tilde{A}, \tilde{b})$ represents an implicit method, often a Diagonally Implicit RK (DIRK), used for the stiff part G in (1.8) [1, 2, 14]. The compact form of (1.8) is written as, with the following definition $\mathbb{K} = \begin{bmatrix} A & 0 \\ b^T & 0 \end{bmatrix}, \tilde{\mathbb{K}} = \begin{bmatrix} \tilde{A} & 0 \\ \tilde{b}^T & 0 \end{bmatrix}$

$$Y = \mathbf{e} \otimes u_n + h(\mathbb{K} \otimes I)F + h(\tilde{\mathbb{K}} \otimes I)G \tag{1.9}$$

Where $\otimes$ denotes the Kronecker product. Notice (1.9) is the same as [9, equation 1.7].

Additive IMEX RK methods have been demonstrated to improve the efficiency of numerical simulations in various applications. Most often, the methods are constructed with the aim of optimizing the *region of absolute monotonicity* (AM) for the explicit method, maximizing the SSP coefficients $\mathcal{C}$, while enforcing L-Stability for the implicit scheme. In [10], the authors demonstrated a more efficient second order scheme with additional properties which "promise reliable and efficient simulations" at the cost of reducing the SSP coefficient from the optimum. In [10] the design philosophy that is described focuses on the following properties:

1. IMEX RK scheme should be of at least least second order; error constant should be small;
2. IMEX RK scheme, (A, b) and $(\tilde{A}, \tilde{b})$, should be SSP with $\mathcal{C}, \tilde{\mathcal{C}} > 0$
3. Implicit scheme $(\tilde{A}, \tilde{b})$ should maintain L-Stability; stability region should contain a large subinterval of the negative real axis $[-z, 0], z > 0$
4. Explicit scheme (A, b): stability region should contain a large subinterval of the negative real axis $[-z, 0], z > 0$; and also the imaginary axis $[-iw, iw], w > 0$
5. Both methods: stability function should be nonnegative for a large interval of negative real axis $[-z, 0], z > 0$;
6. Region of AM of the IMEX RK methods should be large

In this study these ideas are extended in search of a higher order IMEX-RK method $p \leq 4$; where the explicit and the implicit are SSP. Two classes of implicit methods were searched: Singly-Diagonally Implicit Runge–Kutta (SDIRK) and Diagonally Implicit Runge–Kutta (DIRK). Often, the coefficients of the IMEX methods are further restricted to rational

numbers. This study does not enforce this restriction. Previous studies [21, 2, 19, 10] only exhibit order conditions for $p \leq 3$ Additive RK (ARK); the fourth order ($p = 4$) conditions are listed in Table 2.1.

The structure of the paper is organized as follows: in Section 2, a short review of SSP theory is presented; the necessary order conditions for $p \leq 4$ are presented in Section 2.1; the relationship between Butcher and Shu-Osher form is presented Section 2.3; the numerical optimization problem is described in Section 2.4. Some newly constructed methods are presented in Section 3; numerical results are presented in section 4; and finally the conclusion is presented in Section 5.

2. Review of known concepts. The SSP coefficient is for most known RK methods not very large; for a broad class of explicit general linear methods, it is never greater than the number of stages of the method [15]. For many classes of implicit methods it is conjectured to be bounded by twice the number of stages [12, 6, 16].

2.1. Order Condition. The order p of an Additive Runge–Kutta method is determined by necessary conditions imposed on its coefficients $a_{ij}, b_i, c_i, \tilde{a}_{ij}, \tilde{b}_i, \tilde{c}_i$. The conditions for $p \leq 4$ are summarized in Table 2.1 with the following additional definition: $C = \text{diag}(c_i), \tilde{C} = \text{diag}(\tilde{c}_i)$, $\mathbf{e} = (1, 1, \cdots, 1) \in \mathbb{R}^s$.

p (#)	order conditions
1 (2)	$\mathbf{b}^T\mathbf{e} = 1,\ \tilde{\mathbf{b}}^T\mathbf{e} = 1$
2 (4)	$\mathbf{b}^T\mathbf{c} = 1/2,\ \tilde{\mathbf{b}}^T\tilde{\mathbf{c}} = 1/2,\ \mathbf{b}^T\tilde{\mathbf{c}} = 1/2,\ \tilde{\mathbf{b}}^T\tilde{\mathbf{c}} = 1/2$
3 (16)	$\tilde{\mathbf{b}}^T A\tilde{\mathbf{c}} = 1/6,\ \tilde{\mathbf{b}}^T A\mathbf{c} = 1/6,\ \tilde{\mathbf{b}}(\mathbf{c}\cdot\tilde{\mathbf{c}}) = 1/3,\ \tilde{\mathbf{b}}(\mathbf{c}\cdot\mathbf{c}) = 1/3$
	$\tilde{\mathbf{b}}^T \tilde{A}\tilde{\mathbf{c}} = 1/6,\ \tilde{\mathbf{b}}^T \tilde{A}\mathbf{c} = 1/6,\ \tilde{\mathbf{b}}(\tilde{\mathbf{c}}\cdot\mathbf{c}) = 1/3,\ \tilde{\mathbf{b}}(\tilde{\mathbf{c}}\cdot\tilde{\mathbf{c}}) = 1/3$
	$\mathbf{b}^T A\mathbf{c} = 1/6,\ \mathbf{b}^T A\tilde{\mathbf{c}} = 1/6,\ \mathbf{b}(\mathbf{c}\cdot\tilde{\mathbf{c}}) = 1/3,\ \mathbf{b}(\mathbf{c}\cdot\mathbf{c}) = 1/3$
	$\mathbf{b}^T \tilde{A}\tilde{\mathbf{c}} = 1/6,\ \mathbf{b}^T \tilde{A}\mathbf{c} = 1/6,\ \mathbf{b}(\tilde{\mathbf{c}}\cdot\mathbf{c}) = 1/3,\ \mathbf{b}(\tilde{\mathbf{c}}\cdot\tilde{\mathbf{c}}) = 1/3$
4 (64)	$\mathbf{b}^T(\mathbf{ccc}) = 1/4,\ \mathbf{b}^T(\mathbf{cc}\tilde{\mathbf{c}}) = 1/4,\ \mathbf{b}^T CA\mathbf{c} = 1/8,\ \mathbf{b}^T CA\tilde{\mathbf{c}} = 1/8$
	$\mathbf{b}^T(\mathbf{c}\tilde{\mathbf{c}}\mathbf{c}) = 1/4,\ \mathbf{b}^T(\mathbf{c}\tilde{\mathbf{c}}\tilde{\mathbf{c}}) = 1/4,\ \mathbf{b}^T C\tilde{A}\mathbf{c} = 1/8,\ \mathbf{b}^T C\tilde{A}\tilde{\mathbf{c}} = 1/8$
	$\mathbf{b}^T(\tilde{\mathbf{c}}\mathbf{cc}) = 1/4,\ \mathbf{b}^T(\tilde{\mathbf{c}}\mathbf{c}\tilde{\mathbf{c}}) = 1/4,\ \mathbf{b}^T \tilde{C}A\mathbf{c} = 1/8,\ \mathbf{b}^T \tilde{C}A\tilde{\mathbf{c}} = 1/8$
	$\mathbf{b}^T(\tilde{\mathbf{c}}\tilde{\mathbf{c}}\mathbf{c}) = 1/4,\ \mathbf{b}^T(\tilde{\mathbf{c}}\tilde{\mathbf{c}}\tilde{\mathbf{c}}) = 1/4,\ \mathbf{b}^T \tilde{C}\tilde{A}\mathbf{c} = 1/8,\ \mathbf{b}^T \tilde{C}\tilde{A}\tilde{\mathbf{c}} = 1/8$
	$\tilde{\mathbf{b}}^T(\mathbf{ccc}) = 1/4,\ \tilde{\mathbf{b}}^T(\mathbf{cc}\tilde{\mathbf{c}}) = 1/4,\ \tilde{\mathbf{b}}^T CA\mathbf{c} = 1/8,\ \tilde{\mathbf{b}}^T CA\tilde{\mathbf{c}} = 1/8$
	$\tilde{\mathbf{b}}^T(\mathbf{c}\tilde{\mathbf{c}}\mathbf{c}) = 1/4,\ \tilde{\mathbf{b}}^T(\mathbf{c}\tilde{\mathbf{c}}\tilde{\mathbf{c}}) = 1/4,\ \tilde{\mathbf{b}}^T C\tilde{A}\mathbf{c} = 1/8,\ \tilde{\mathbf{b}}^T C\tilde{A}\tilde{\mathbf{c}} = 1/8$
	$\tilde{\mathbf{b}}^T(\tilde{\mathbf{c}}\mathbf{cc}) = 1/4,\ \tilde{\mathbf{b}}^T(\tilde{\mathbf{c}}\mathbf{c}\tilde{\mathbf{c}}) = 1/4,\ \tilde{\mathbf{b}}^T \tilde{C}A\mathbf{c} = 1/8,\ \tilde{\mathbf{b}}^T \tilde{C}A\tilde{\mathbf{c}} = 1/8$
	$\tilde{\mathbf{b}}^T(\tilde{\mathbf{c}}\tilde{\mathbf{c}}\mathbf{c}) = 1/4,\ \tilde{\mathbf{b}}^T(\tilde{\mathbf{c}}\tilde{\mathbf{c}}\tilde{\mathbf{c}}) = 1/4,\ \tilde{\mathbf{b}}^T \tilde{C}\tilde{A}\mathbf{c} = 1/8,\ \tilde{\mathbf{b}}^T \tilde{C}\tilde{A}\tilde{\mathbf{c}} = 1/8$
	$\mathbf{b}^T A(\mathbf{cc}) = 1/12,\ \mathbf{b}^T A(\mathbf{c}\tilde{\mathbf{c}}) = 1/12,\ \mathbf{b}^T(AA)\mathbf{c} = 1/24,\ \mathbf{b}^T(AA)\tilde{\mathbf{c}} = 1/24$
	$\mathbf{b}^T A(\tilde{\mathbf{c}}\mathbf{c}) = 1/12,\ \mathbf{b}^T A(\tilde{\mathbf{c}}\tilde{\mathbf{c}}) = 1/12,\ \mathbf{b}^T(A\tilde{A})\mathbf{c} = 1/24,\ \mathbf{b}^T(A\tilde{A})\tilde{\mathbf{c}} = 1/24$
	$\mathbf{b}^T \tilde{A}(\mathbf{cc}) = 1/12,\ \mathbf{b}^T \tilde{A}(\mathbf{c}\tilde{\mathbf{c}}) = 1/12,\ \mathbf{b}^T(\tilde{A}A)\mathbf{c} = 1/24,\ \mathbf{b}^T(\tilde{A}A)\tilde{\mathbf{c}} = 1/24$
	$\mathbf{b}^T \tilde{A}(\tilde{\mathbf{c}}\mathbf{c}) = 1/12,\ \mathbf{b}^T \tilde{A}(\tilde{\mathbf{c}}\tilde{\mathbf{c}}) = 1/12,\ \mathbf{b}^T(\tilde{A}\tilde{A})\mathbf{c} = 1/24,\ \mathbf{b}^T(\tilde{A}\tilde{A})\tilde{\mathbf{c}} = 1/24$
	$\tilde{\mathbf{b}}^T A(\mathbf{cc}) = 1/12,\ \tilde{\mathbf{b}}^T A(\mathbf{c}\tilde{\mathbf{c}}) = 1/12,\ \tilde{\mathbf{b}}^T(AA)\mathbf{c} = 1/24,\ \tilde{\mathbf{b}}^T(AA)\tilde{\mathbf{c}} = 1/24$
	$\tilde{\mathbf{b}}^T A(\tilde{\mathbf{c}}\mathbf{c}) = 1/12,\ \tilde{\mathbf{b}}^T A(\tilde{\mathbf{c}}\tilde{\mathbf{c}}) = 1/12,\ \tilde{\mathbf{b}}^T(A\tilde{A})\mathbf{c} = 1/24,\ \tilde{\mathbf{b}}^T(A\tilde{A})\tilde{\mathbf{c}} = 1/24$
	$\tilde{\mathbf{b}}^T \tilde{A}(\mathbf{cc}) = 1/12,\ \tilde{\mathbf{b}}^T \tilde{A}(\mathbf{c}\tilde{\mathbf{c}}) = 1/12,\ \tilde{\mathbf{b}}^T(\tilde{A}A)\mathbf{c} = 1/24,\ \tilde{\mathbf{b}}^T(\tilde{A}A)\tilde{\mathbf{c}} = 1/24$
	$\tilde{\mathbf{b}}^T \tilde{A}(\tilde{\mathbf{c}}\mathbf{c}) = 1/12,\ \tilde{\mathbf{b}}^T \tilde{A}(\tilde{\mathbf{c}}\tilde{\mathbf{c}}) = 1/12,\ \tilde{\mathbf{b}}^T(\tilde{A}\tilde{A})\mathbf{c} = 1/24,\ \tilde{\mathbf{b}}^T(\tilde{A}\tilde{A})\tilde{\mathbf{c}} = 1/24$

Table 2.1: Order conditions of Additive IMEX Runge–Kutta methods for $p \leq 4$.

Due to the complexity of the necessary order conditions that must be satisfied to obtain higher order methods, it is common to make assumptions which reduce the number of equations. Under each assumption, we obtain different classes of modified methods: *Type-G* refer to methods which make no assumption on the coefficients of the scheme (Table 2.1); *Type-B* refer to methods which assume that $\mathbf{b} = \tilde{\mathbf{b}}$ (Table 2.2); and *Type-BC* will refer to methods for which the coefficients $\mathbf{b} = \tilde{\mathbf{b}}$ and $\mathbf{c} = \tilde{\mathbf{c}}$ (Table 2.3). After elimination of redundancy, a similar order condition for $p = 3$ of *Type-G*, can be found in [2]. Furthermore, we will refer to methods as follows: SSP$q(s,r,p)$-GBCLSM, where the letters have the following meaning [10]:

q: order the explicit scheme
s: number of stages of explicit scheme
r: number of stages of implicit scheme
p: order of the IMEX scheme
'G': *Type-G* method
'B': *Type-B* method i.e $\mathbf{b} = \tilde{\mathbf{b}}$
'C': *Type-BC* method i.e $\mathbf{b} = \tilde{\mathbf{b}}$ and $\mathbf{c} = \tilde{\mathbf{c}}$
'L': L-stable
'S': stability of the explicit part contains an interval on the imaginary axis
'M': the IMEX method has a nontrivial region of absolute monotonicity

p (#)	order conditions
1 (1)	$\mathbf{b}^T\mathbf{e} = 1$
2 (2)	$\mathbf{b}^T\mathbf{c} = 1/2$ $\mathbf{b}^T\tilde{\mathbf{c}} = 1/2$
3 (7)	$\mathbf{b}^T A\mathbf{c} = 1/6,\ \mathbf{b}^T A\tilde{\mathbf{c}} = 1/6,\ \mathbf{b}(\mathbf{c}\cdot\tilde{\mathbf{c}}) = 1/3,\ \mathbf{b}(\mathbf{c}\cdot\mathbf{c}) = 1/3$ $\mathbf{b}^T \tilde{A}\tilde{\mathbf{c}} = 1/6,\ \mathbf{b}^T \tilde{A}\mathbf{c} = 1/6,\ \mathbf{b}(\tilde{\mathbf{c}}\cdot\tilde{\mathbf{c}}) = 1/3$
4 (26)	$\mathbf{b}^T(\mathbf{ccc}) = 1/4$ $\mathbf{b}^T(\mathbf{cc}\tilde{\mathbf{c}}) = 1/4,\ \mathbf{b}^T CA\mathbf{c} = 1/8,\ \mathbf{b}^T CA\tilde{\mathbf{c}} = 1/8$ $\mathbf{b}^T(\mathbf{c}\tilde{\mathbf{c}}\tilde{\mathbf{c}}) = 1/4,\ \mathbf{b}^T C\tilde{A}\mathbf{c} = 1/8,\ \mathbf{b}^T C\tilde{A}\tilde{\mathbf{c}} = 1/8$ $\mathbf{b}^T \tilde{C}A\mathbf{c} = 1/8$ $\mathbf{b}^T \tilde{C}A\tilde{\mathbf{c}} = 1/8$ $\mathbf{b}^T(\tilde{\mathbf{c}}\tilde{\mathbf{c}}\tilde{\mathbf{c}}) = 1/4,\ \mathbf{b}^T \tilde{C}\tilde{A}\mathbf{c} = 1/8,\ \mathbf{b}^T \tilde{C}\tilde{A}\tilde{\mathbf{c}} = 1/8$ $\mathbf{b}^T A(\mathbf{cc}) = 1/12,\ \mathbf{b}^T A(\mathbf{c}\tilde{\mathbf{c}}) = 1/12,\ \mathbf{b}^T(AA)\mathbf{c} = 1/24,\ \mathbf{b}^T(AA)\tilde{\mathbf{c}} = 1/24$ $\mathbf{b}^T A(\tilde{\mathbf{c}}\tilde{\mathbf{c}}) = 1/12,\ \mathbf{b}^T(A\tilde{A})\mathbf{c} = 1/24,\ \mathbf{b}^T(A\tilde{A})\tilde{\mathbf{c}} = 1/24,\ \mathbf{b}^T \tilde{A}(\mathbf{cc}) = 1/12$ $\mathbf{b}^T \tilde{A}(\mathbf{c}\tilde{\mathbf{c}}) = 1/12,\ \mathbf{b}^T(\tilde{A}A)\mathbf{c} = 1/24,\ \mathbf{b}^T(\tilde{A}A)\tilde{\mathbf{c}} = 1/24,\ \mathbf{b}^T \tilde{A}(\tilde{\mathbf{c}}\tilde{\mathbf{c}}) = 1/12$ $\mathbf{b}^T(\tilde{A}\tilde{A})\mathbf{c} = 1/24,\ \mathbf{b}^T(\tilde{A}\tilde{A})\tilde{\mathbf{c}} = 1/24$

Table 2.2: Reduced order conditions ($\mathbf{b} = \tilde{\mathbf{b}}$) for $p \leq 4$.

Methods developed in [10] correspond to *Type-B* under our classification. Similar assumptions were made in [14] to make *Type-BC* methods attainable.

2.2. Radius and Regions of Absolute Monotonicity. For RK and IMEX RK, the step size restrictions to obtain SSP or TVD methods are given, respectively, by the Kraaijevanger radius and the region of absolute monotonicity.

Definition 2.1 ([9] Definition 2.1, [17] Definition 2.4). *An s-stage RK method with coefficient $\mathbb{K}$ is said to be absolutely monotonic (a.m) at a given point $\xi \leq 0$ if the matrix $I - \xi\mathbb{K}$ is nonsingular and*

$$(I - \xi\mathbb{K})^{-1}\mathbb{K} \geq 0 \tag{2.1}$$

$$(I - \xi\mathbb{K})^{-1}e \geq 0 \tag{2.2}$$

p	order conditions
1 (1)	$\mathbf{b}^T\mathbf{e} = 1,\ \ \mathbf{c} = \tilde{\mathbf{c}}$
2 (1)	$\mathbf{b}^T\mathbf{c} = 1/2$
3 (3)	$\mathbf{b}^T A\mathbf{c} = 1/6,\ \ \mathbf{b}^T \tilde{A}\mathbf{c} = 1/6,\ \ \mathbf{b}^T\mathbf{c}^2 = 1/3$
4 (9)	$\mathbf{b}^T\mathbf{c}^3 = 1/4,\ \ \mathbf{b}^T CA\mathbf{c} = 1/8,\ \ \mathbf{b}^T C\tilde{A}\mathbf{c} = 1/8,$ $\mathbf{b}^T A\mathbf{c}^2 = 1/12,\ \ \mathbf{b}^T (AA)\mathbf{c} = 1/24,\ \ \mathbf{b}^T (A\tilde{A})\mathbf{c} = 1/24$ $\mathbf{b}^T \tilde{A}\mathbf{c}^2 = 1/12,\ \ \mathbf{b}^T (\tilde{A}A)\mathbf{c} = 1/24,\ \ \mathbf{b}^T (\tilde{A}\tilde{A})\mathbf{c} = 1/24$

Table 2.3: Reduced order conditions ($\mathbf{b} = \tilde{\mathbf{b}}, \mathbf{c} = \tilde{\mathbf{c}}$) for $p \leq 4$.

where $e = (1, 1, \cdots, 1)^t \in \mathbb{R}^{s+1}$, and the vector inequalities are understood componentwise. Further, the method is said to a.m. on a given set $\Omega \subset \mathbb{R}$ if it is a.m. at each $\xi \in \Omega$. The radius of absolute monotonicity $R(\mathbb{K})$ is defined by

$$R(\mathbb{K}) = \sup\{r | r \geq 0 \text{ and } \mathbb{K} \text{ is a.m on } [-r, 0]\} \tag{2.3}$$

if there is no $r > 0$ such that $\mathbb{K}$ is a.m, on $[-r, 0]$, we set $R(\mathbb{K}) = 0$.

Definition 2.2 ([9] Defintion 2.2). *An s-stage ARK IMEX method $(\mathbb{K}, \tilde{\mathbb{K}})$ is said to be a.m at a given point $(\xi, \tilde{\xi})$ with $\xi, \tilde{\xi} \leq 0$ if the matrix $I - \xi\mathbb{K} - \tilde{\xi}\tilde{\mathbb{K}}$ is invertible and*

$$A(\xi, \tilde{\xi}) = (I - \xi\mathbb{K} - \tilde{\xi}\tilde{\mathbb{K}})^{-1}\mathbb{K} \geq 0 \tag{2.4}$$
$$\tilde{A}(\xi, \tilde{\xi}) = (I - \xi\mathbb{K} - \tilde{\xi}\tilde{\mathbb{K}})^{-1}\tilde{\mathbb{K}} \geq 0 \tag{2.5}$$
$$e(\xi, \tilde{\xi}) = (I - \xi\mathbb{K} - \tilde{\xi}\tilde{\mathbb{K}})^{-1}e \geq 0 \tag{2.6}$$

further, the additive method is said to be a.m on a given set $\Omega \in \mathbb{R}^2$ if it is absolutely monotonic at each $(\xi, \tilde{\xi}) \in \Omega$

Definition 2.3 (Definition 2.3 [9]). *The region of absolute monotonicity, denoted by $\mathcal{R}(\mathbb{K}, \tilde{\mathbb{K}})$ is defined by*

$$\mathcal{R}(\mathbb{K}, \tilde{\mathbb{K}}) = \{(r, \tilde{r}) | r \geq 0, \tilde{r} \geq 0, \text{ and } (\mathbb{K}, \tilde{\mathbb{K}}) \text{ is a.m. on } [-r, 0] \times [-\tilde{r}, 0]\} \tag{2.7}$$

In [9, Proposition 2.11], the authors extended the results of [18, Lemma 4.4], and demonstrated that to prove the monotonicity of an ARK method in the segment that connects the origin and the point $(-r, -\tilde{r})$, it is enough to check the absolute monotonicity of the method $(\mathbb{K}, \tilde{\mathbb{K}})$ at $(-r, -\tilde{r})$, and the nonnegativity of $\mathbb{K}$ and $\tilde{\mathbb{K}}$.

Thus, the new methods are constructed such that the method $(\mathbb{K}, \tilde{\mathbb{K}}) \geq 0$ satisfy the order conditions listed in Section 2.1 and are absolutely monotonic at $(-r, -\tilde{r})$.

2.3. ARK methods in Shu-Osher representation. The derivation of monotonicity is conceptually much easier when the ARK method is written in Shu-Osher form [9, 7]. The canonical Shu-Osher representation of (1.8):

$$\begin{aligned} \mathbf{y}^{(i)} &= v_i\mathbf{y}^{(n)} + \sum_{j=1}^{s}\left(\alpha_{ij}\mathbf{y}^{(j)} + \Delta t\beta_{ij}F(\mathbf{y}^{(j)})\right) + \sum_{j=1}^{s}\left(\tilde{\alpha}_{ij}\mathbf{y}^{(j)} + \Delta t\tilde{\beta}_{ij}G(\mathbf{y}^{(j)})\right) \\ \mathbf{y}^{(n+1)} &= \mathbf{y}^{(s+1)} \end{aligned} \tag{2.8}$$

For consistency, require that

$$1 = \sum_{j=1}^{s} \alpha_{ij} + \sum_{j=1}^{s} \tilde{\alpha}_{ij}, \quad 1 \le i \le s+1 \tag{2.9}$$

The relation between the Shu-Osher form and the Butcher form is as follow:

$$(\alpha)_{ij} = \begin{cases} \alpha_{ij} & 1 \le i \le s+1, \quad 1 \le j \le s \\ 0 & j = s+1 \end{cases} \quad (\beta)_{ij} = \begin{cases} \beta_{ij} & 1 \le i \le s+1, \quad 1 \le j \le s \\ 0 & j = s+1 \end{cases}$$

$$(\tilde{\alpha})_{ij} = \begin{cases} \tilde{\alpha}_{ij} & 1 \le i \le s+1, \quad 1 \le j \le s \\ 0 & j = s+1 \end{cases} \quad \left(\tilde{\beta}\right)_{ij} = \begin{cases} \tilde{\beta}_{ij} & 1 \le i \le s+1, \quad 1 \le j \le s \\ 0 & j = s+1 \end{cases}$$

$$\mathbf{Y} = \bar{\mathbf{v}}\mathbf{y}^{(n)} + \boldsymbol{\alpha}\mathbf{Y} + \Delta t \boldsymbol{\beta} F + \tilde{\boldsymbol{\alpha}}\mathbf{Y} + \Delta t \tilde{\boldsymbol{\beta}} G, \qquad \mathbf{y}^{(n+1)} = \mathbf{Y}^{s+1} \tag{2.10}$$

where $\bar{\mathbf{v}} = I \otimes v, \boldsymbol{\alpha} = I \otimes \alpha, \tilde{\boldsymbol{\alpha}} = I \otimes \tilde{\alpha}, \boldsymbol{\beta} = I \otimes \beta, \tilde{\boldsymbol{\beta}} = I \otimes \tilde{\beta}$. Taking $\boldsymbol{\alpha}, \tilde{\boldsymbol{\alpha}} = 0$ in Modified Shu-Osher Form (2.10), and $\mathbb{K} = \beta_{ij}, \tilde{\mathbb{K}} = \tilde{\beta}_{ij}, \bar{\mathbf{v}} = \mathbf{e}$

$$\mathbf{Y} = \mathbf{e}u^n + \Delta t \mathbb{K} F + \Delta t \tilde{\mathbb{K}} G \tag{2.11}$$

In general, $\mathbb{K} = (I - \boldsymbol{\alpha} - \tilde{\boldsymbol{\alpha}})^{-1}\boldsymbol{\beta}, \tilde{\mathbb{K}} = (I - \boldsymbol{\alpha} - \tilde{\boldsymbol{\alpha}})^{-1}\tilde{\boldsymbol{\beta}}$ and $(\boldsymbol{\alpha} + \tilde{\boldsymbol{\alpha}})\mathbf{e} + \bar{\mathbf{v}} = \mathbf{e}$

Definition 2.4 ([7] Zero-well-defined). *A Runge–Kutta method is zero-well-defined if the stage equations have a unique solution when the method is applied to the scalar initial value problem*

$$\boldsymbol{y}'(t) = 0, \qquad \boldsymbol{y}(t_0) = \boldsymbol{y}_0.$$

We prove the monotonicity property (1.4) using the Shu-Osher representation (2.8)

$$\mathbf{y}^{(i)} = v_i \mathbf{y}^{(n)} + \sum_{j=1}^{s} + \left(\alpha_{ij}\mathbf{y}^{(j)} + \Delta t \beta_{ij} F(\mathbf{y}^{(j)})\right) + \sum_{j=1}^{s} \left(\tilde{\alpha}_{ij}\mathbf{y}^{(j)} + \Delta t \tilde{\beta}_{ij} G(\mathbf{y}^{(j)})\right)$$

We know $\alpha_{ij}, \tilde{\alpha}_{ij} \ge 0$ and from the consistency condition (2.9), $1 - \sum_{j=1}^{s} \alpha_{ij} - \sum_{j=1}^{s} \tilde{\alpha}_{ij} = v_i$

$$\|\mathbf{y}^{(i)}\| \le (v_i)\, \|\mathbf{y}^{(n)}\| + \sum_{j=1}^{s} \alpha_{ij} \| \left(\mathbf{y}^{(j)} + \Delta t \frac{\beta_{ij}}{\alpha_{ij}} F(\mathbf{y}^{(j)})\right) \| + \sum_{j=1}^{s} \tilde{\alpha}_{ij} \| \left(\mathbf{y}^{(j)} + \Delta t \frac{\tilde{\beta}_{ij}}{\tilde{\alpha}_{ij}} G(\mathbf{y}^{(j)})\right) \|$$

$$\|\mathbf{y}^{(i)}\| \le \left(1 - \sum_{j=1}^{s} \alpha_{ij} - \sum_{j=1}^{s} \tilde{\alpha}_{ij}\right) \|\mathbf{y}^{(n)}\| + \sum_{j=1}^{s} \alpha_{ij} \|\mathbf{y}^{(j)}\| + \sum_{j=1}^{s} \tilde{\alpha}_{ij} \|\mathbf{y}^{(j)}\| \qquad 1 \le i \le s+1 \tag{2.12}$$

$$\| \left(\mathbf{y}^{(j)} + \Delta t \tfrac{\beta_{ij}}{\alpha_{ij}} F(\mathbf{y}^{(j)})\right) \| \le \|\mathbf{y}^{(j)}\|, \qquad \| \left(\mathbf{y}^{(j)} + \Delta t \tfrac{\tilde{\beta}_{ij}}{\tilde{\alpha}_{ij}} G(\mathbf{y}^{(j)})\right) \| \le \|\mathbf{y}^{(j)}\|$$

Just as in the SSP Book [7], let q be the index of the ARK stage with the largest norm, i.e $q \in \{1, 2, \cdots s+1\}$ such that $\|\mathbf{y}^{(i)}\| \le \|\mathbf{y}^{(q)}\|$ for all $1 \le i \le s+1$. Taking $i = q$:

$$\|\mathbf{y}^{(q)}\| \leq \left(1 - \sum_{j=1}^{s} \alpha_{qj} - \sum_{j=1}^{s} \tilde{\alpha}_{qj}\right) \|\mathbf{y}^{(n)}\| + \sum_{j=1}^{s} \alpha_{qj} \|\mathbf{y}^{(j)}\| + \sum_{j=1}^{s} \tilde{\alpha}_{qj} \|\mathbf{y}^{(j)}\|$$

$$\|\mathbf{y}^{(j)}\| \leq \|\mathbf{y}^{(q)}\| \qquad \forall 1 \leq j \leq s+1$$

$$\|\mathbf{y}^{(q)}\| \leq \left(1 - \sum_{j=1}^{s} \alpha_{qj} - \sum_{j=1}^{s} \tilde{\alpha}_{qj}\right) \|\mathbf{y}^{(n)}\| + \sum_{j=1}^{s} \alpha_{qj} \|\mathbf{y}^{(q)}\| + \sum_{j=1}^{s} \tilde{\alpha}_{qj} \|\mathbf{y}^{(q)}\|$$

$$\left(1 - \sum_{j=1}^{s} \alpha_{qj} - \sum_{j=1}^{s} \tilde{\alpha}_{qj}\right) \|\mathbf{y}^{(q)}\| \leq \left(1 - \sum_{j=1}^{s} \alpha_{qj} - \sum_{j=1}^{s} \tilde{\alpha}_{qj}\right) \|\mathbf{y}^{(n)}\|$$

$$\left(1 - \sum_{j=1}^{s} \alpha_{qj} - \sum_{j=1}^{s} \tilde{\alpha}_{qj}\right) \neq 0 \rightarrow \|\mathbf{y}^{(q)}\| \leq \|\mathbf{y}^{(n)}\|$$

suppose instead $\left(1 - \sum_{j=1}^{s} \alpha_{qj} - \sum_{j=1}^{s} \tilde{\alpha}_{qj}\right) = 0$, then

$$\|\mathbf{y}^{(q)}\| \leq \left(1 - \sum_{j=1}^{s} \alpha_{qj} - \sum_{j=1}^{s} \tilde{\alpha}_{qj}\right) \|\mathbf{y}^{(n)}\| + \sum_{j=1}^{s} \alpha_{ij} \|\mathbf{y}^{(j)}\| + \sum_{ij} \tilde{\alpha}_{ij} \|\mathbf{y}^{(j)}\|$$

$$\|\mathbf{y}^{(i)}\| \leq \|\mathbf{y}^{(q)}\|, \qquad \forall 1 \leq i \leq s+1$$

$$\|\mathbf{y}^{(q)}\| \leq \sum_{j=1}^{s} \alpha_{ij} \|\mathbf{y}^{(j)}\| + \sum_{j=1}^{s} \tilde{\alpha}_{ij} \|\mathbf{y}^{(j)}\| \rightarrow \|\mathbf{y}^{(q)}\| \leq \left(\sum_{j=1}^{s} \alpha_{ij} + \sum_{j=1}^{s} \tilde{\alpha}_{ij}\right) \|\mathbf{y}^{(j)}\|$$

recall that q is chosen so that $\|\mathbf{y}^{(i)}\| \leq \|\mathbf{y}^{(q)}\|$, this implied that $\|\mathbf{y}^{(j)}\| = \|\mathbf{y}^{(q)}\|$ for every j such that $\left(\sum_{j=1}^{s} \alpha_{ij} + \sum_{j=1}^{s} \tilde{\alpha}_{ij}\right) \neq 0$. Let $J = \{j : \left(\sum_{j=1}^{s} \alpha_{ij} + \sum_{j=1}^{s} \tilde{\alpha}_{ij}\right) \neq 0\}$. If there exist $j^* \in J$ such that $1 - \left(\sum_{j=1}^{s} \alpha_{ij} + \sum_{j=1}^{s} \tilde{\alpha}_{ij}\right) \neq 0$, it is easy to verify that the method is zero-well-defined by $q = j^*$. If there is no $j^* \in J$ such that $1 - \left(\sum_{j=1}^{s} \alpha_{ij} + \sum_{j=1}^{s} \tilde{\alpha}_{ij}\right) \neq 0$, then it follows that the stages with the indices in J depends only on each other and not on $\mathbf{y}^{(n)}$. Thus the method is not zero-well-defined [7], completing the proof.

2.4. Numerical Optimization. Large classes of RK methods found in literature were obtained by numerical optimization [7, Sec. 3.4]. The construction of new IMEX RK methods is posed as a numerical optimization problem:

$$\begin{aligned} &\underset{\mathcal{C},\tilde{\mathcal{C}}}{\text{maximize}} \\ &\text{subject to} \quad A(\mathcal{C},\tilde{\mathcal{C}}) \geq 0, \quad \tilde{A}(\mathcal{C},\tilde{\mathcal{C}}) \geq 0, \; e(\mathcal{C},\tilde{\mathcal{C}}) \geq 0, \quad \tau_p(A,b,\tilde{A},\tilde{b}) = 0 \end{aligned} \tag{2.13}$$

where $A(\mathcal{C},\tilde{\mathcal{C}}), \tilde{A}(\mathcal{C},\tilde{\mathcal{C}}), e(\mathcal{C},\tilde{\mathcal{C}})$ are defined in (2.4), (2.5), (2.6) respectively; and τ_p represents the set of necessary order conditions for order p, see Table 2.2.

Moreover, the following restrictions on the stability functions are required:

$$r(z) = 1 + zb^T(I - zA)^{-1}e \qquad \tilde{r}(z) = 1 + z\tilde{b}^T(I - z\tilde{A})^{-1}e \tag{2.14}$$

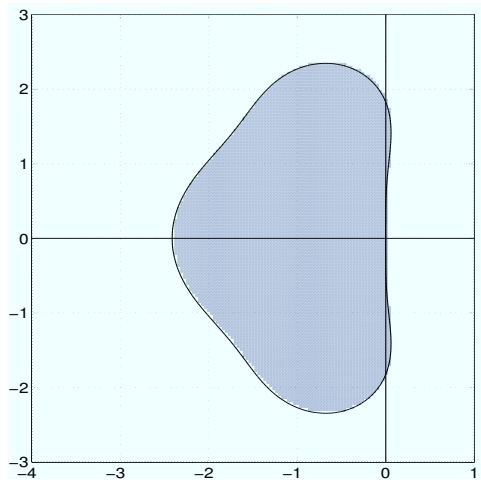
- $|r(iw)| \leq 1, w > 0$: intersection with the imaginary axis
- $|r(-w)| \leq 1, w > 0$: intersection of the stability region with the real axis

- $r(-w) \geq 0, w > 0$: non-negativity of the stability region
- $\lim_{z\to\infty} \tilde{r}(z) = 0$: enforcing L-Stability for an A-Stable scheme

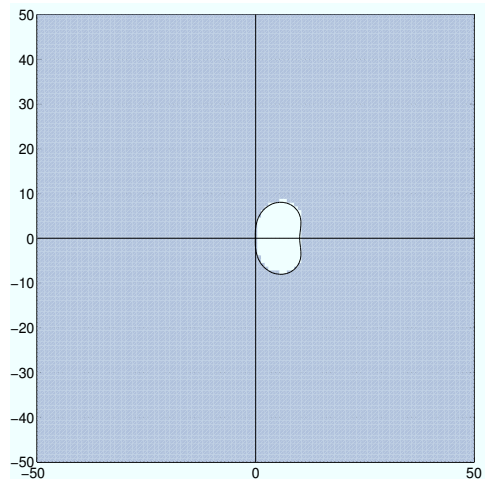
The above optimization problem was implemented in MATLAB using the sequential quadratic programming (SQP) algorithm (*fmincon* in the optimization toolbox) to numerically search for higher order methods with nontrivial SSP coefficient.

3. New Schemes. This Section presents some of the new constructed methods ; each properly marked with the stability and properties it inherits. Recall that L-Stable implicit schemes $(\tilde{A}, \tilde{b})$ are required. Although *Type-BC* have reduced set of necessary order-conditions, the numerical optimization did not converge to an IMEX method of this type with an implicit part satisfying the L-Stability requirement. We believe this is due to singularity of the coefficient matrix $\tilde{A}$ when the condition $\mathbf{c} = \tilde{\mathbf{c}}$ is enforced, further consideration of this result is ongoing.

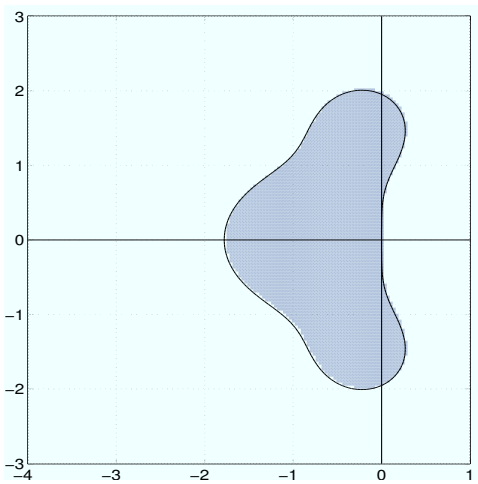
The new methods were obtained from the numerical optimization formulated in Section 2.4. As evident from the stability plots, the optimized explicit method $\mathbb{K}$ contains nontrivial subinterval of the imaginary axis, large subinterval of the negative real axis. Other classes of methods, such as *Type-BC* were found but are not represented here; in general these methods have less desirable stability regions.



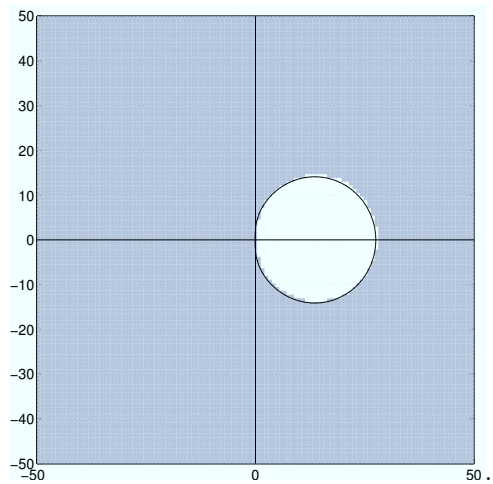
(c) Explicit SSP2(3,3,2)-BLSM



(d) Implicit SSP2(3,3,2)-BLSM

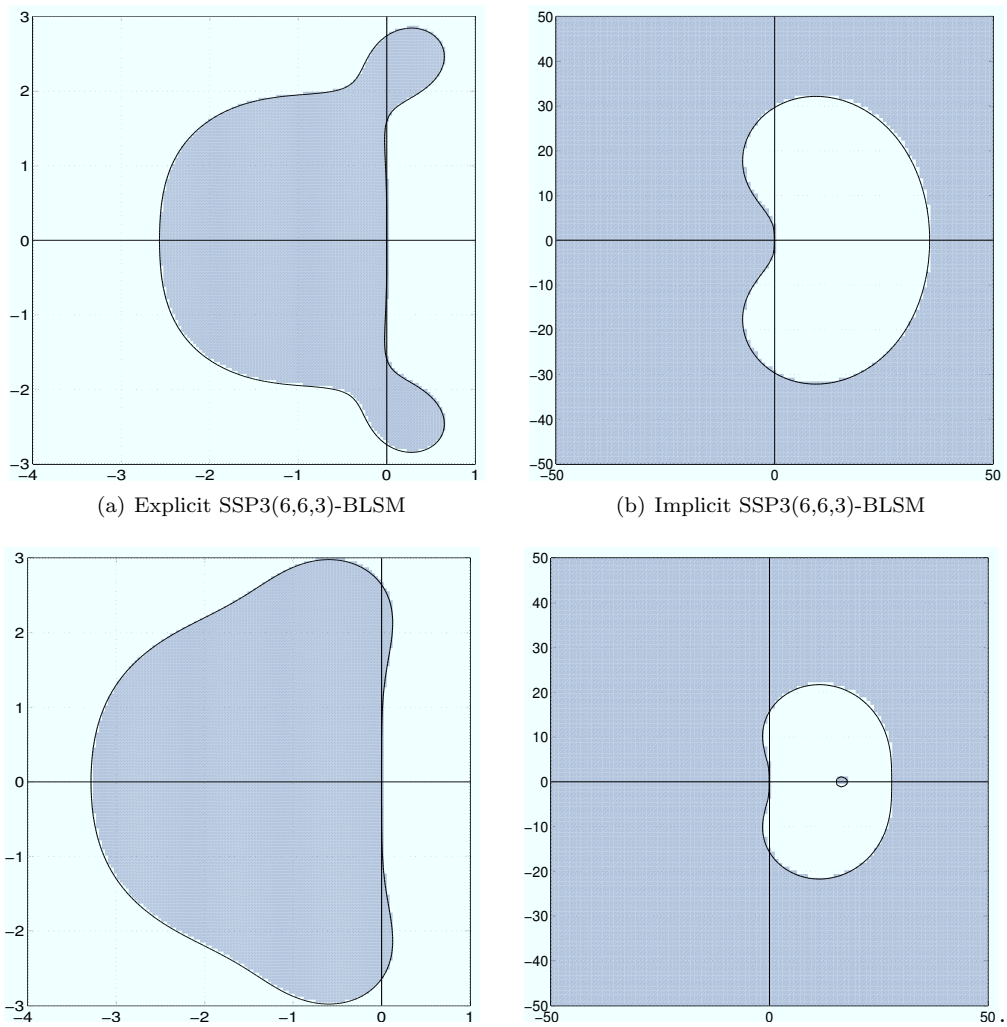


(e) Explicit SSP2(4,4,2)-BLSM



(f) Implicit SSP2(4,4,2)-BLSM

Fig. 3.1: Stability region (shaded areas) of explicit method (Figure 3.1(c)) and implicit method (Figure 3.1(d)) of SSP2(3,3,2)-BLSM: $\mathcal{C} = 1.3, \tilde{\mathcal{C}} = 2.6$; explicit method (Figure 3.1(e)) and implicit method (Figure 3.1(f)) of SSP2(4,4,2)-BLSM: $\mathcal{C} = 1.3, \tilde{\mathcal{C}} = 2.4$. Note that the explicit method contains nontrivial subinterval of the imaginary and the negative real axis



(a) Explicit SSP3(6,6,3)-BLSM

(b) Implicit SSP3(6,6,3)-BLSM

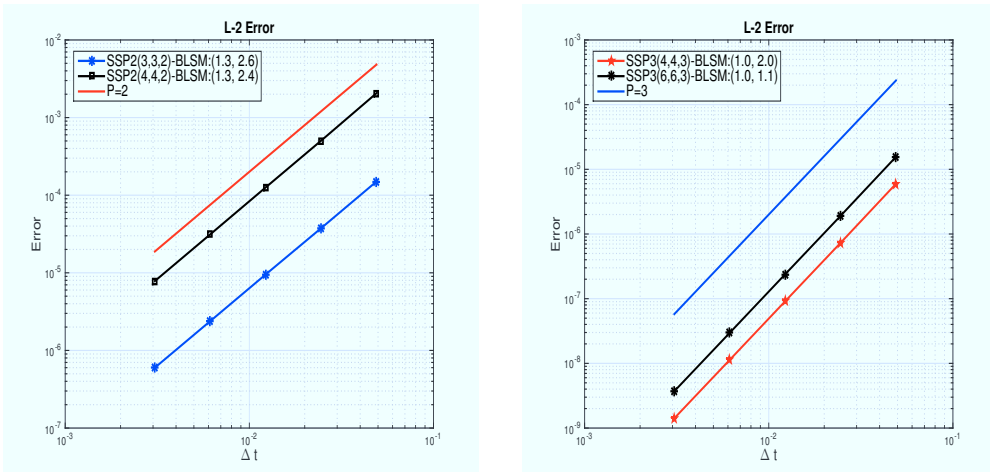
(c) Explicit SSP3(4,4,3)-BLSM

(d) Implicit SSP3(4,4,3)-BLSM

Fig. 3.2: Stability region of explicit method (Figure 3.2(a)) and implicit method (Figure 3.2(b)) of SSP3(6,6,3)-BLSM: $\mathcal{C} = 1.0, \tilde{\mathcal{C}} = 1.1$; explicit method (Figure 3.2(c)) and implicit method (Figure 3.2(d)) of SSP3(4,4,3)-BLSM: $\mathcal{C} = 1.0, \tilde{\mathcal{C}} = 2.0$. Note that the explicit method contains nontrivial subinterval of the imaginary and the negative real axis.

4. Numerical Results.

4.1. Order Verification. In this test, the aim is to verify the convergence (order-of-accuracy) of the SSP IMEX RK method on an advection-diffusion PDE $\frac{\partial u}{\partial t} + a\frac{\partial u}{\partial x} = \nu\frac{\partial^2 u}{\partial x^2}$ with the initial condition $u(x, t = 0) = A\sin(2\pi kx)$ and known exact exact solution $u(x, t) = A\exp(-\nu k^2 t)\sin(2\pi k(x - at))$.



(a) Second Order (b) Third Order

Fig. 4.1: Convergence result of the methods, using spectral method with 64 spatial points and $k = 1, a = 1, \nu = 0.5$

Figure 4.1 shows that the methods obtained the expected formal order-of-accuracy.

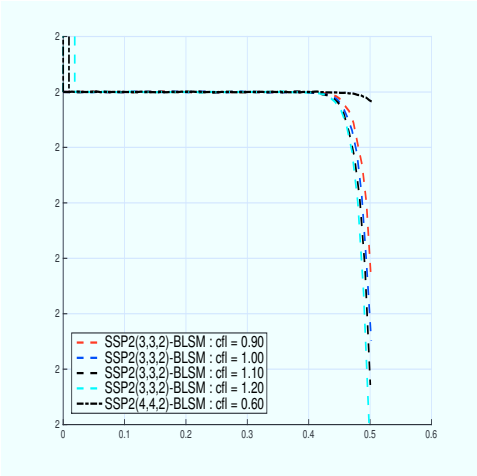
4.2. SSP Study. In Section 4.1, it was shown that the methods attained the correct order of accuracy. In this section, the aim is to show the SSP property of the methods.

As a model problem a coupled-first order hyperbolic problem is solved for both the explicit ($\mathbb{K}$) and the implicit ($\tilde{\mathbb{K}}$) Runge–Kutta to show that both methods do maintain the desired Total Variation (TV) Diminishing stability. For this simple test, $a = b = 1$ from (4.1), which results in the CFL simply being the observed SSP coefficient.

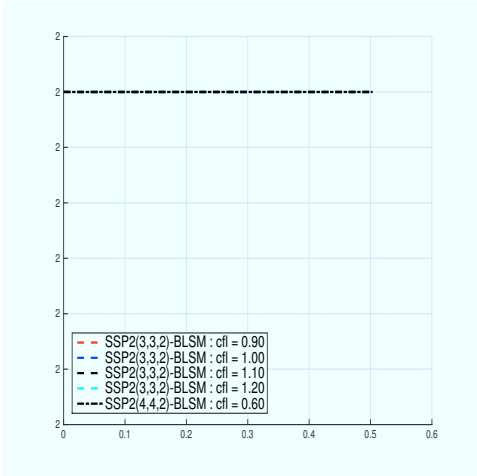
$$u_t + av_x = 0 \qquad v_t + bu_x = 0 \tag{4.1}$$

To do this, the first order upwind discretization in space is employed, which is known to be TVD [8]. The initial condition is a square wave for both $v(x, t)$ and $u(x, t)$.

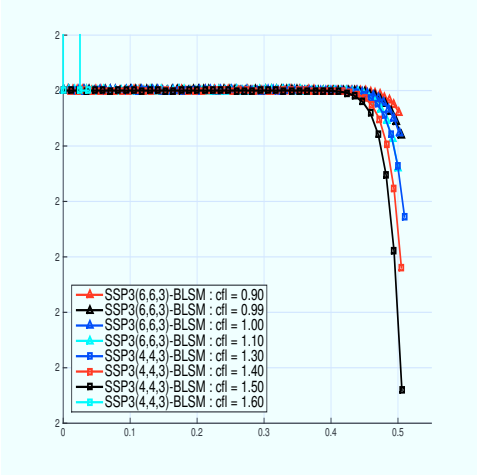
Figures 4.2 show the observed total variation of Equation (4.1) for $p = 2$ and $p = 3$. Figure 4.2(a) shows that the explicit solution is TVD for $CFL < 1.20$; an overshoot occurs in the very first step in the solution for $CFL = 1.20$. In Figure 4.2(c), SSP3(4,4,3)-BLSM violates TVD for the explicit method at $CFL = 1.60 > \mathcal{C}$ in the very first time step $TVD \approx 2.072491 > 2.0$. There are no TVD violation in the implicit solution since $CFL < \tilde{\mathcal{C}}$.



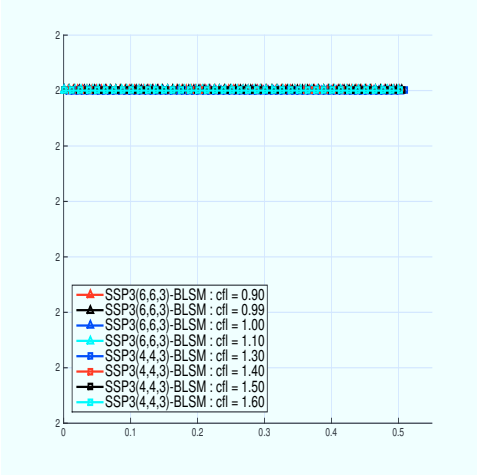
(a) TV($u(x,t)$)



(b) TV($v(x,t)$)



(c) TV($u(x,t)$)



(d) TV($v(x,t)$)

Fig. 4.2: Figure 4.2(a) (explicit) and Figure 4.2(b) (implicit) shows the total variation of the square waves for SSP2(3,3,2)-BLSM.and SSP2(4,4,2)-BLSM Figure 4.2(c) (explicit) and Figure 4.2(d) (implicit) shows the total variation of the square waves for SSP3(4,4,3)-BLSM and SSP3(6,6,3)-BLSM .

5. Conclusion. Through the paper, the construction of the new IMEX RK was demonstrated through the identification of the properties necessary to achieve an accurate and stable numerical simulation. The demonstrated methods are overall SSP IMEX RK of order $p \leq 3$, with nontrivial region AM ($\mathcal{C}, \tilde{\mathcal{C}} > 0$); the explicit method containing imaginary axis $[-iw, iw], w > 0$; the implicit method maintaining the L-Stability; and both methods both containing a large subinterval of the negative real axis $[-z, 0], z > 0$.

In order to achieve a 4th order scheme for the Runge–Kutta method, a Type-BC assumption was made, however, such methods were unable to maintain L-Stability, which is

crucial to the stiffness of the implicit part. Although accurate, due to the lack of L-Stability, these methods are not recommended for practical use. The Type-BC assumption enforces structure on previously stated methods; the conjecture follows that structure makes it impossible for a fourth order Type-BC method to maintain L-stability. A formal proof of the conjecture and more numerical examples will be pursued in the future work.

REFERENCES

[1] Uri M. Ascher, Steven J. Ruuth, and Raymond J. Spiteri. Implicit–Explicit Runge–Kutta Methods for Time-dependent Partial Differential Equations. *Appl. Numer. Math*, 25:151–167, 1997.

[2] S. Boscarino, L. Pareschi, and G. Russo. Implicit–Explicit Runge–Kutta Schemes for Hyperbolic Systems and Kinetic Equations in the Diffusion Limit. *SIAM Journal on Scientific Computing*, 35(1):A22–A51, 2013.

[3] Sebastiano Boscarino. Error Analysis of Imex Runge-Kutta Methods Derived from Differential-Algebraic Systems. *SIAM Journal on Numerical Analysis*, 45(4):1600–1621, 2007.

[4] Kevin Burrage and R.P.K. Chan. On Smoothing and Order Reduction Effects for Implicit Runge–Kutta Formulae, 1993.

[5] John C Butcher. On Runge–Kutta processes of high order. *Journal of the Australian Mathematical Society*, 4(02):179–194, 1964.

[6] L. Ferracina and M. N. Spijker. Strong Stability of Singly–Diagonally–Implicit Runge–Kutta Methods. *Appl. Numer. Math.*, 58(11):1675–1686, November 2008.

[7] S. Gottlieb, D.I. Ketcheson, and C.W. Shu. *Strong Stability Preserving Runge–Kutta and Multistep Time Discretizations*. World Scientific, 2011.

[8] Sigal Gottlieb and Chi-Wang Shu. Total Variation Diminishing Runge–Kutta Schemes. *Mathematics of Computation*, 67(221):73–86, January 1998.

[9] Inmaculada Higueras. Strong Stability for Additive Runge-Kutta Methods. *SIAM Journal on Numerical Analysis*, 44(4):1735–1758, 2006.

[10] Inmaculada Higueras, Natalie Happenhofer, Othmar Koch, and Friedrich Kupka. Optimized Strong Stability Preserving IMEX Runge–Kutta Methods. *J. Computational Applied Mathematics*, 272:116–140, 2014.

[11] W. Hundsdorfer and J.G. Verwer. *Numerical Solution of Time-Dependent Advection-Diffusion-Reaction Equations*. Springer Series in Computational Mathematics. Springer, 2003.

[12] Willem Hundsdorfer and Steven J. Ruuth. On Monotonicity and boundedness properties of linear multistep methods. *J. MATH. COMPUT.*, 75(254):655–672, April 2006.

[13] Willem Hundsdorfer and M. N. Spijker. Boundedness and Strong Stability of Runge–Kutta Methods. *Math. Comput.*, 80(274):863–886, 2011.

[14] Christopher A. Kennedy and Mark H. Carpenter. Additive Runge–Kutta Schemes for Convection–Diffusion–Reaction Equations. *Appl. Numer. Math.*, 44(1-2):139–181, January 2003.

[15] David I. Ketcheson. Computation of Optimal Monotonicity Preserving General Linear Methods. *Mathematics of Computation*, July 2009.

[16] David I. Ketcheson, Colin B. Macdonald, and Sigal Gottlieb. Optimal Implicit Strong Stability Preserving Runge–Kutta Methods. *Appl. Numer. Math.*, 59(2):373–392, February 2009.

[17] J.F.B.M. Kraaijevanger. Contractivity of Runge–Kutta Methods. *BIT Numerical Mathematics*, 31(3):482–528, 1991.

[18] J.F.B.M. Kraaijevanger. Contractivity of Runge–Kutta Methods. *BIT Numerical Mathematics*, 31(3):482–528, 1991.

[19] Lorenzo Pareschi and Giovanni Russo. Implicit-Explicit Runge-Kutta Schemes and Applications to Hyperbolic Systems with Relaxation. *Journal of Scientific Computing*, 25(1):129–155, 2005.

[20] Joachim Rang. An Analysis of the Prothero-Robinson Example for Constructing New Adaptive ESDIRK Methods of Order 3 and 4. *Appl. Numer. Math.*, 94(C):75–87, August 2015.

[21] Russell Williams, Kevin Burrage, Ian Cameron, and Minnie Kerr. A Four-stage Index 2 Diagonally Implicit Runge-Kutta Method. *Applied Numerical Mathematics*, 40(3):415 – 432, 2002.

PERIDYNAMIC MULTISCALE FINITE ELEMENT METHOD

TIMOTHY B. COSTA*, STEPHEN D. BOND†, DAVID LITTLEWOOD‡, AND STAN MOORE§

Abstract. In this work we present a Nonlocal Multiscale Finite Element Method which solves the peridynamic model at multiple scales to include microscale information at the coarse-scale. We then consider a method that solves a fine-scale peridynamic model to build element-support basis functions for a coarse-scale local partial differential equation model, called the Mixed Locality Multiscale Finite Element Method.

1. Introduction. The problem of computing quantum-accurate design-scale solutions to mechanics problems is rich with applications and serves as the background to modern multiscale science research. The problem can be broken into component problems comprised of communicating across adjacent scales, which when strung together create a pipeline for information to travel between quantum scales and design-scales. Traditionally, this involves connections between a) quantum electronic structure calculations and molecular dynamics and between b) molecular dynamics and local partial differential equation models at the design-scale. The second step, b), is particularly challenging since the appropriate scales of molecular dynamic and local partial differential equation models do not overlap. The peridynamic model for continuum mechanics provides a solution to this problem, as the basic equations of peridynamics are mathematically valid at a wide range of scales limiting from the classical partial differential equation models valid at the design-scale to the scale of molecular dynamics. In this work we focus on the development of multiscale finite element methods for the peridynamic model, in an effort to create a mathematically consistent channel for microscale information to travel from the upper limits of the molecular dynamics scale to the design-scale. In particular, we first develop a Nonlocal Multiscale Finite Element Method which solves the peridynamic model at multiple scales to include microscale information at the coarse-scale. We then consider a method that solves a fine-scale peridynamic model to build element-support basis functions for a coarse-scale local partial differential equation model, calld the Mixed Locality Multiscale Finite Element Method. Additionally, we present a novel Galerkin framework, the 'Ambulant Galerkin Method', which represents a first step towards a unified mathematical analysis of local and nonlocal multiscale finite element methods, and whose future extension will allow the analysis of multiscale finite element methods that mix models across scales under certain assumptions of the consistency of those models.

This results in this report, and additional results, are given in detail in [4]. This report is organized as follows. In Section 2 we present a Galerkin framework, called the Ambulant Galerkin Method, which will serve as the background to the mathematical analysis of the Nl-MSFEM and future extensions. In Section 3 we present the peridynamic model of solid mechanics. In Section 4 we present the nonlocal multiscale finite element method, as well as a variant that connects nonlocal and local models across scales. In Section 6 we present numerical results. Finally in Section 7 we give concluding remarks.

2. Ambulant Galerkin Method. In this section we present a Galerkin framework called the 'Ambulant Galerkin Method.' This method is motivated by a desire for an abstract framework for the analysis of multiscale finite element methods designed for local, nonlocal, or mixed-locality problems. The principal idea behind the method is the construction of a

*Department of Mathematics, Oregon State University, costat@math.oregonstate.edu
†Center for Computing Research, Sandia National Laboratories, sdbond@sandia.gov
‡Center for Computing Research, Sandia National Laboratories, djlittl@sandia.gov
§Center for Computing Research, Sandia National Laboratories, stamoor@sandia.gov

correction operator which translates an approximating subspace toward the true solution in order to reduce error without increasing the dimension of the approximating subspace.

Let V be a Hilbert space, $V' = \mathcal{L}(V, \mathbb{R})$ be the space of continuous linear functionals on V. Let $\mathcal{B} \in \mathcal{L}(V, V')$ be V-coercive and let $f \in V'$. We consider the abstract problem

$$\text{find } u \in V \ \text{ s.t. } \ \mathcal{B}u - f \in V^o. \tag{2.1}$$

Here the superscript o denotes the annhililator of the space V,

$$V^o := \{f \in V' \ : \ f(v) = 0, \ \forall v \in V\}. \tag{2.2}$$

Our first step is to define an approximation space. Presumably this space has a low number of degrees of freedom, and produces an insufficiently accurate solution with standard methods. Let $V^H \subset V$ be a finite dimensional subspace with basis $\{\phi_i\}_{i=1}^{N_H}$, called the 'approximation space' and let $I^H : V \to V^H$ be an orthogonal projection operator.

In order to decompose the space V into a direct sum decomposition of V^H and a remainder space we take advantage of the properties of an orthogonal projection operator. Define the 'remainder space' V^r by

$$V^r = \{v \in V \ : \ I^H(v) = 0\}, \tag{2.3}$$

so that

$$V = V^H \oplus V^r. \tag{2.4}$$

Our goal now is to enhance the space V^H without increasing the total degrees of freedom in the approximation problem, i.e. without increasing the dimension of the approximation space V^H. To do this, we will define a correction operator that takes advantage of the direct sum decomposition $V = V^H \oplus V^r$ to translate the space V^H within V to a space of the same dimension, which contains the exact solution. This correction operator $Q : V^H \to V^r$ is defined as the solution to the following problem:

$$\text{given } \phi \in V^H, \text{ find } Q(\phi) \in V^r \ \text{ s.t. } \ \mathcal{B}(\phi + Q(\phi)) - f \in (V^r)^o. \tag{2.5}$$

The correction operator acts on a function $\phi \in V$ and produces a function $Q(\phi) \in V^r$ such that $\phi + Q(\phi)$ now contains information about the solution to the problem posed on V (2.1). To make this precise, define the reconstruction operator $R : V^H \to V$ as $R = Id + Q$, then define the 'ambulant' space

$$V^A = \text{span}\{R(\phi_i) | \{\phi_i\}_{i=1}^{N_H} \text{ is a basis for } V^H\}. \tag{2.6}$$

As we will show shortly, the ambulant space contains the true solution to the model problem (2.1). To see this, consider the Petrov-Galerkin problem where we use the ambulant space as the trial space and the original approximation space V^H as the test space:

$$\text{find } u^A \in V^A \ : \ \mathcal{B}u^A - f \in (V^H)^o. \tag{2.7}$$

Lemma 2.1. *(Proof in [4]) Problems* (2.1) *and* (2.7) *are equivalent.*

Additionally, we can state the standard Galerkin problem on V^A and obtain a similar result:

$$\text{find } u \in V^A \ \text{ s.t. } \ \mathcal{B}u - f \in (V^A)^o. \tag{2.8}$$

Lemma 2.2. *(Proof in [4]) Problems* (2.1) *and* (2.8) *are equivalent.*

2.1. Ambulant Approximations. Lemmas 2.1 and 2.2 point out that while (2.7) and (2.8) are formally finite dimensional, the dependence of Q on the dimension of V^r results in infinite dimensional problems. In this section we consider finite dimensional approximations of the operator Q and the convergence of the corresponding methods in the Galerkin and Petrov-Galerkin settings.

To design finite dimensional approximations, we introduce a family of finite dimensional spaces $\{V^a \mid a > 0\}$ such that

$$V \supset V^a \supset V^H, \quad \forall a. \tag{2.9}$$

To assist with the convergence analysis we make a sensible assumption on the family of spaces $\{V^a\}$,

TCASSUMPTION 1. *The family $\{V^a\}_a$ is dense in V in the sense that for each $v \in V$ there exists a sequence $\{v_n \in V^{a_n}\}$ with $a_n \to 0$ as $n \to \infty$ such that*

$$\|v - v_n\|_V \to 0 \text{ as } n \to \infty. \tag{2.10}$$

Choosing a particular a, we then we set

$$V^{r,a} = Ker(I^H|_{V^a}). \tag{2.11}$$

Then we define the ambulant approximation space V^{A_a} by the basis functions $\{R^a(\phi_i)\}_{i=1}^{N_H}$ where $R^a = Id + Q^a$ and $Q^a(\phi_i)$ is the solution to the problem,

$$\text{find } Q^a(\phi_i) \in V^{r,a} \quad \text{s.t.} \quad \mathcal{B}(\phi_i + Q^a(\phi_i)) - f \in (V^{r,a})^o. \tag{2.12}$$

2.1.1. Petrov-Galerkin Formulation. With the finite dimensional approximation operator Q^a at hand we can define the Ambulant Petrov-Galerkin problem,

$$\text{find } u^{pg} \in V^{A_a} \quad \text{s.t.} \quad \mathcal{B}u^{pg} - f \in (V^H)^o. \tag{2.13}$$

We would like to know that the Ambulant Petrov-Galerkin method converges to the true solution as $a \to 0$.

THEOREM 2.3. *(Proof in [4]) Under Assumption 1 we have,*

$$\lim_{a \to 0} \|u - u^{pg}\|_V = 0. \tag{2.14}$$

So, we see that without changing the approximation space V^H, we can converge to the true solution with refinement only in the residual space $V^{r,a}$.

2.1.2. Galerkin Formulation. Next we consider the finite dimensional Ambulant Galerkin problem,

$$\text{find } u^g \in V^{A_a} \quad \text{s.t.} \quad \mathcal{B}u^g - f \in (V^{A_a})^o. \tag{2.15}$$

THEOREM 2.4. *(Proof in [4]) Under Assumption 1 we have,*

$$\lim_{a \to 0} \|u - u^g\|_V = 0. \tag{2.16}$$

2.2. Source Removal. In this section we analyze the error induced by removing the source term f from the calculation of the ambulant basis functions in the abstract framework. Thus we replace (2.12) with

$$\text{find } Q_*^a(\phi_i) \in V^{r,a} \text{ s.t. } \mathcal{B}(\phi_i + Q_*^a(\phi_i)) \in (V^{r,a})^o. \tag{2.17}$$

There are several reasons one may be interested in removing the source term. First, as we will see, in the multiscale finite element context we are interested in obtaining basis functions that correspond to material heterogeneity, not to larger scale force variations. Removing the source term encourages the computed basis functions to respond solely to the material parameters within $\mathcal{B}$. Additionally, we have the following result stating that with no source the reconstruction operator is an orthogonality preserving map.

PROPOSITION 2.5. *Applying the Riesz Representation Theorem, write $\mathcal{B}u(v) = (\mathcal{B}u, v)_V$, so that in* (2.17) *$\mathcal{B} : V^{r,a} \to V^{r,a}$. Assume $\mathcal{B}$ is a self-adjoint operator. Define $R_*^a = Id + Q_*^a$ where Q_*^a is given by* (2.17)*. Then R_*^a preserves orthogonality. In particular, if $\{\phi_i\}_{i=1}^{N_H}$ is an orthogonal basis for V^H, then $\{R_*^a(\phi_i)\}_{i=1}^{N_H}$ is an orthogonal basis for V^{A_a}.*

Let α be the coercivity constant for the operator $\mathcal{B}$, and consider the difference,

$$\begin{aligned}\alpha\|R_*^a(\phi_i) - R^a(\phi_i)\|_V^2 &\leq (\mathcal{B}(R_*^a(\phi_i) - R^a(\phi_i)), R_*^a(\phi_i) - R^a(\phi_i))_V \\ &= (\mathcal{B}R_*^a(\phi_i), R_*^a(\phi_i))_V - (\mathcal{B}R_*^a(\phi_i), R^a(\phi_i))_V \\ &\quad - (\mathcal{B}R^a(\phi_i), R_*^a(\phi_i))_V + (\mathcal{B}R^a(\phi_i), R^a(\phi_i))_V. \end{aligned} \tag{2.18}$$

After a technical calculation and applying (2.12) and (2.17) we have,

$$\alpha\|R_*^a(\phi_i) - R^a(\phi_i)\|_V^2 \leq (f, Q^a(\phi_i) - Q_*^a(\phi_i))_V. \tag{2.19}$$

This tells us that in the finite element context, the error induced by dropping the source term will be on the order of the mesh size in V^H.

3. Peridynamic Model of Continuum Mechanics. The peridynamic theory of continuum mechanics is a nonlocal theory that remains valid in the presence of discontinuities. The theory has several advantages over the traditional theory, including natural modeling of material failure, validity across a wide range of scales, and the ability to model nonlocal effects.

The original peridynamic theory (bond-based peridynamics) was introduced by S.A. Silling in 2000 [21]. In 2007 [18] Silling et al. introduced the state-based peridynamic model, which significantly enhanced the types of materials the theory is able to model. Significant work has been done on comparing the peridynamic model to discrete models and classical continuum theory. In 2008 Silling and Lehoucq [25] established, under suitable assumptions, the convergence of the peridynamic model to classical elasticity theory in the limit of vanishing nonlocality. Lehcoucq and Silling, and Seleson [11, 16] examined the relationship between molecular dynamics and the peridynamic theory, and it showed that molecular dynamics is a special case of the peridynamic model when generalized functions are used in the constitutive model. In 2015 [13] Rahman and Foster examined the relationship between statistical mechanics and the peridynamic theory.

Let $D \subset \mathbb{R}^d$, $d \in \{1, 2, 3\}$, denote a material body. The principal assumption in the peridynamic theory is that any body-point x in the reference configuration is acted upon by forces due to the deformation of all the body-points q within some neighborhood of finite radius δ. The radius δ is referred to as the *horizon*, and the set of points within this neighborhood is referred to as the *family* of x, typically denoted $\mathcal{H}_x = \{q \in D \mid |q - x| < \delta\}$.

In the peridynamic theory, a concise form of the equation of motion is stated as

$$\rho(x)u_{tt}(x,t) = \int_{\mathcal{H}_x} f(q,x,t)\ dq + b(x,t), \quad x \in D,\ t > 0, \tag{3.1}$$

where $\rho(x)$ is the density of the body in a reference configuration, $u(x,t)$ is the displacement field, and b is a prescribed body force density. The first term on the right hand side is the internal force density, and it is here that we see, mathematically, the expression of nonlocality. The internal force density at a point x depends on all points in the family of x. Further, for a given body-point $x \in D$ it is assumed that beyond the horizon, q exerts no force on x. So,

$$|q - x| > \delta \Rightarrow f(q,x,t) = 0. \tag{3.2}$$

The function f contains information about the deformation and the material model, both from the point x and the point q.

In this work we restrict ourselves to a linear peridynamic body, for which (3.1) has the form,

$$\rho(x)u_{tt}(x,t) = \int_D C(x,q)(u(q,t) - u(x,t))\ dq + b(x,t), \tag{3.3}$$

where $C(\cdot,\cdot) : D \times D \to \mathbb{R}^{d\times d}$ is a tensor-valued micro-modulus function which describes the material. We note that the linear peridynamic model assumes small displacements, but does not prohibit fracture. We require that the micromodulus function $C(x,q)$ is symmetric in its arguments so that the integrand is anti-symmetric in accordance with Newton's third law.

Additionally, the peristatic problem has a similar formulation,

$$\int_D C(x,q)(u(q) - u(x))\ dq + b(x,t) = 0. \tag{3.4}$$

3.1. Weak Formulation of Peridynamics. As has been well-established at this point, standard boundary conditions are insufficient for the nonlocal peridynamic model. Instead, we require a volume constraint. To introduce the volume constraint, we define $D_{\mathcal{I}}$, the interaction domain, as

$$D_{\mathcal{I}} = \{x \notin D\ :\ \mathrm{dist}(x, \partial D) \le \delta\}, \tag{3.5}$$

and

$$D' = D \cup D_{\mathcal{I}}. \tag{3.6}$$

Then we define the bilinear forms,

$$(\cdot,\cdot)_\rho : V \times V \to \mathbb{R}\ :\ (u,v)_\rho = \int_{D'} \rho uv\ dx, \tag{3.7}$$

$$(\cdot,\cdot) : V \times V \to \mathbb{R}\ :\ (u,v) = \int_{D'} uv\ dx, \tag{3.8}$$

$$B : V \times V \to \mathbb{R}\ :\ B(u,v) = \frac{1}{2}\int_{D'}\int_{D'} C(x,q)(u(q) - u(x))(v(q) - v(x))\ dq\ dx. \tag{3.9}$$

The energy space associated with the linear peridynamic problem is given by

$$\mathcal{E} = \{w : D' \to \mathbb{R}^d \ : \ B(w,w) < \infty\}. \tag{3.10}$$

with the semi-norm,

$$|w|_{\mathcal{E}} = (B(w,w))^{\frac{1}{2}} \tag{3.11}$$

Then choosing the homogeneous Dirichlet problem, we define

$$V = \{w \in \mathcal{E} \ : \ w|_{D_{\mathcal{I}}} = 0\}, \tag{3.12}$$

with the inner product and norm,

$$(u,v)_V = B(u,v), \tag{3.13}$$

$$\|u\|_V^2 = B(u,u). \tag{3.14}$$

Finally, we define,

$$W = H^2(0,T;V) := \left\{ v : [0,T] \to V \ \middle| \ \left(\int_0^T \|\partial_t^i v(t)\|_V^2 \, dt \right)^{\frac{1}{2}} < \infty, \ \ i = 0,1,2 \right\}. \tag{3.15}$$

Then we have the weak form of the homogeneous Dirichlet linear peridynamic problem,

$$\text{find } u \in W \ : \ (u_{tt}(t), v)_\rho + B(u(t), v) = (b(t), v), \quad \forall v \in V \text{ and a.e. } t \in [0,T]. \tag{3.16}$$

And for the static case,

$$\text{find } u \in V \ : \ B(u,v) = (b,v), \quad \forall v \in V. \tag{3.17}$$

4. Peridynamic Multiscale Finite Element Method. In this section we describe how the AFEM framework connects directly to a nonlocal multiscale finite element method for the peristatic problem (3.17). We then consider a method that solves a fine-scale peridynamic model to build element-support basis functions for a coarse-scale local partial differential equation model, called the Mixed Locality Multiscale Finite Element Method (ML-MSFEM).

4.1. Nonlocal Multiscale Finite Element Method. Let $\mathcal{T}^H = \{T_i\}_{i=1}^N$ be a regular triangulation of the domain D' into intervals ($d = 1$), triangles ($d = 2$), or tetrahedrons ($d = 3$). Here H is the mesh size, and we consider a discontinuous piece-wise linear finite element space. We note that a discontinuous Galerkin approximation to the peridynamic problem is conforming [3]. Denote by M the number of basis functions on each element. Let $\{\phi_{i,j}\}$, $i \in \{1,\dots,N\}$ and $j \in \{1,\dots,M\}$ be the basis for the finite element problem, so that $V^H = \text{span}\{\phi_{i,j}\}_{ij}$.

For the space V^a we take the simplest case of a regular mesh refinement of the mesh $\mathcal{T}^H$. Denoting this new mesh by $\mathcal{T}^a$, we have $\mathcal{T}^a = \{Y_{i,k}\}_{ik}$. Throughout, we will use i to refer to a coarse element in $\mathcal{T}^H$, j to index coarse basis functions $\phi_{i,j}$ on each coarse element, and $k \in \{1,\dots,N_i\}$ to index fine elements in the mesh refinement within coarse element i. In this context, a refers to the mesh size of the refined mesh $\mathcal{T}^a$. Finally, on each coarse element T_i, and each fine element $Y_{i,k}$ within T_i, we again define a discontinuous piecewise linear approximation through basis functions $\{\psi_{i,k,l}\}_{ikl}$. Here $l \in \{1,\dots,M\}$ indexes basis

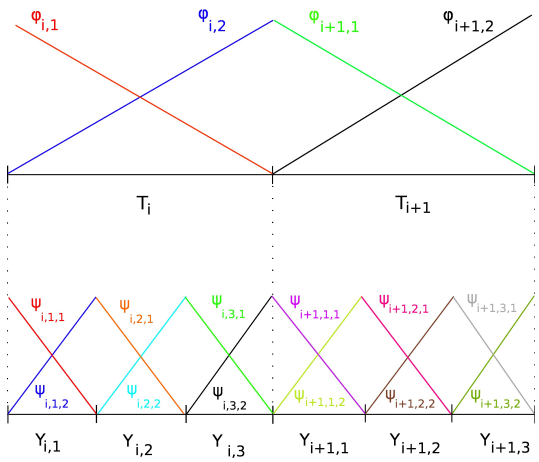


Fig. 4.1: Illustration of element and basis indexing in 1D.

functions in the finite element space V^a defined on element $Y_{i,k}$. Figure 4.1 illustrates this setup in 1D. In this setup we have $N_H = M * N$ and $N_a = M \sum_{i=1}^{N} N_i$.

Then for each pair (i, j) the ambulant basis function $R_*^a(\phi_{i,j}) = \phi_{i,j} + Q_*^a(\phi_{i,j})$ is computed by,

$$B(Q_*^a(\phi_{i,j}), \psi_{i,k,l}) = -B(\phi_{i,j}, \psi_{i^*,k,l}) \quad i^* = 1, \ldots, N,\ k = 1, \ldots, N_i,\ l = 1, \ldots, M. \tag{4.1}$$

Notice that in (4.1) the computations are performed on the entire domain D', and we expect that $R_*^a(\phi_{i,j})$ will have support outside of T_i.

To obtain a method that computes $R_*^a(\phi_{i,j})$ locally, we replace (4.1) with

$$B(Q_*^a(\phi_{i,j}), \psi_{i,k,l}) = -B(\phi_{i,j}, \psi_{i,k,l}) \quad k = 1, \ldots, N_i,\ l = 1, \ldots, M. \tag{4.2}$$

We note that the domain T_i for each computation is padded and homogeneous Dirichlet volume constraints are enforced in the padded domain. Then,

$$R_*^a(\phi_{i,j}) = \begin{cases} \phi_{i,j} + Q_*^a(\phi_{i,j}) & x \in T_i \\ 0 & x \notin T_i \end{cases}. \tag{4.3}$$

This method now defines a nonlocal multiscale finite element method.

5. Mixed Locality Finite Element Method. Due to the relatively high cost of solving the weak peridynamic model compared to finite element methods for local models, as well as the decades of work that has gone into the development of FEM codes for local mechanics models, a natural question to ask is, can we use the multiscale finite element method to communicate across scales from nonlocal models (fine-scale) to (coarse-scale) local models of continuum mechanics? We call this method the Mixed Locality Multiscale Finite Element Method (ML-MSFEM), and describe it in this section for a linear elastic material.

The local model corresponding to the linear peristatic problem (3.17) is the linear Poisson equation,

$$\text{find } u \in H_0^1(D) \ \text{ s.t. } \ B^{loc}(u, v) = (b, v), \quad \forall v \in H_0^1(D). \tag{5.1}$$

Here

$$H_0^1 = \{v \in L^2(D) \ : \ \partial_i u \in L^2(D), \ i \in \{x, y, z\}, \ \gamma u = 0\}, \tag{5.2}$$

where ∂_i refers to the distributional derivative in the i direction and γ is the trace operator. Additionally,

$$H^{-1} = (H_0^1)', \tag{5.3}$$

and

$$B^{loc} : H_0^1(D) \times H_0^1(D) \to \mathbb{R}; \quad B^{loc}(u, v) = \int_D \nabla u \cdot \nabla v \ dx. \tag{5.4}$$

The micromodulus function $C(x, q)$ and the coefficient function $\alpha(x)$ are related by

$$C(x, q) = \frac{\alpha(x) + \alpha(q)}{2}. \tag{5.5}$$

For details regarding the assumptions and scaling necessary to obtain this local model as the 0 horizon limit of the linear peridynamic model, the reader is referred to [25]. In fact, we will ultimately be interested in coupling local and nonlocal models for which this convergence does not hold due to, e.g. fracture.

Then the ML-MSFEM method is composed of two steps. First, for each pair (i, j) the multiscale basis function $R_*^a(\phi_{i,j})$ is computed as in the Nl-MSFEM, by (4.2).

The second step requires a choice. The basis functions $\{R_*^a(\phi_{i,j})\}$ are basis functions for a discontinuous Galerkin space (DG), which is non-conforming for the local PDE model (5.1). One could then solve (5.1) with a DG method. Alternatively, we can paste the DG basis functions together across edges to create a basis for a continuous Galerkin space. In this paper we do the latter. Analysis of these options and the method in general will be the subject of later work.

6. Numerical Results. In this section we present numerical results for both the nonlocal and mixed-locality multiscale finite element methods.

6.1. Nonlocal Multiscale Finite Elements. In Figure 6.1 we compare results from the standard multiscale finite element method for the linear Poisson equation in 1D,

$$-(\alpha(x)u_x)_x = b, \quad x \in (0, 1), \tag{6.1}$$

$$u(0) = u(1) = 0, \tag{6.2}$$

and the nonlocal multiscale finite element method for the corresponding peridynamic problem with horizon $\delta = 0.01$. We set

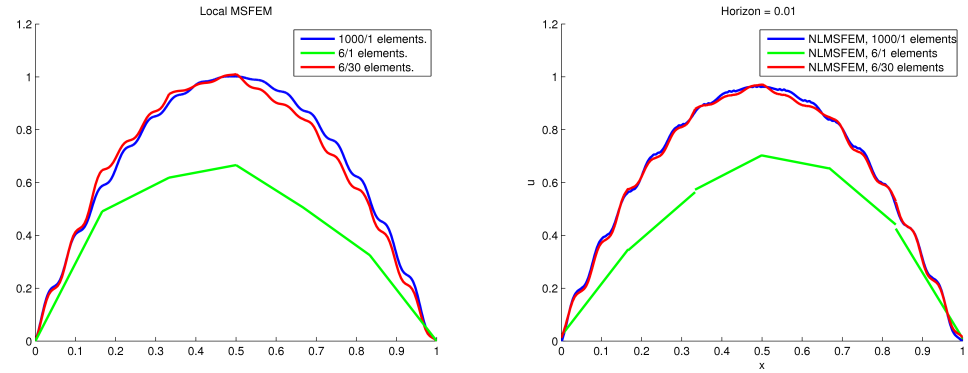
$$\alpha(x) = (4 + 3\sin(100x))^{-1} \tag{6.3}$$

in the local problem, and

$$C(x, q) = \frac{\alpha(x) + \alpha(q)}{2} \tag{6.4}$$

in the nonlocal problem. For both problems the force density is set to

$$b = 1. \tag{6.5}$$



(a) Local multiscale finite element examples. (b) Nonlocal multiscale finite element examples.

Fig. 6.1: Comparison of local and nonlocal multiscale finite element solutions.

In Figure 6.1 for the local (left) and nonlocal (right) problem we compare the solutions obtained by: (a) standard FEM with 1000 elements, (b) standard FEM with 6 elements, and (c) multiscale FEM with 6 coarse elements and 30 fine elements per coarse element. As expected we see good agreement between the solution on the 6 element multiscale space and the 1000 element standard FEM space in both cases, while the standard FEM solution with 6 elements is very poor.

The three solutions displayed in Figure 6.1 also provide a clear basis for reviewing the advantage of the multiscale finite element method in the context of large-scale next generation computing platforms. Table 6.1 lists the degrees of freedom in the global solve for each solution as well as the mesh size H. This shows that the MSFEM framework provides an accurate solution with an extremely small global system to solve and a very large mesh size. This is done by shifting the computational burden to the decoupled fine-scale problems which produce basis functions. These subscale problems can be computed concurrently, with no communication required between nodes, a clear advantage given the current trends in large scale computing. For the stationary problem the advantages here are clear: a smaller global problem to solve reduces solution time and memory requirements. For dynamic problems there is an additional advantage; for any time stepping scheme that is not fully implicit stability requirements put limitations on the size of the time step taken that are a function of the mesh size. For dynamic problems the MSFEM framework allows larger time steps for explicit and semi-implicit schemes by obtaining accuracy in space through the fine-scale solutions while the time step is determined by the global mesh size.

Method	1000/1	6/1	6/30
Global Degrees of Freedom	2000	12	12
Mesh size H	0.001	0.166	0.166.

Table 6.1: Degrees of freedom and mesh size for solutions in Figure 6.1.

Next we perform convergence tests for the nonlocal multiscale finite element method.

In these tests we used the following,

$$u^{exact}(x) = -2x^2 + \frac{x\sin(30\pi x)}{50} + \frac{\cos(30\pi x)}{1500\pi} + 2, \tag{6.6}$$

$$\alpha(x) = \frac{2}{4 - \frac{3}{5}\pi\cos(30\pi x)}. \tag{6.7}$$

In Figure 6.2 we show the convergence as the mesh size H is refined for various fixed number of fine-scale elements used to produced each multiscale basis function on each coarse element. We see that the standard finite element method corresponding to 1 fine-scale element per coarse element fails to achieve the expected convergence rates due to the microscale heterogeneity of the true solution. In contrast, in the Nl-MSFEM tests, for each number of fine-scale elements used to compute multiscale basis functions on the coarse mesh we see order 2 convergence. Additionally, we see a drop in the error as the number of fine elements used to compute the multiscale basis functions is increased.

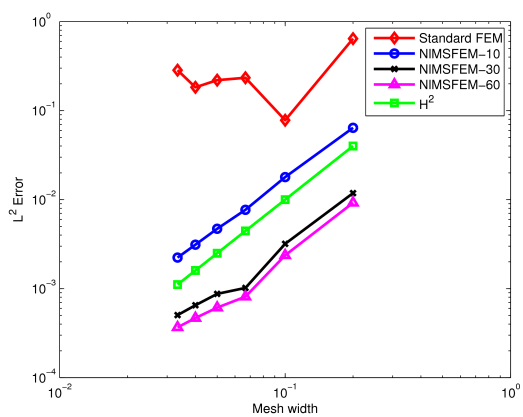


Fig. 6.2: H-convergence for fixed numbers of fine-scale elements used in computing multiscale basis functions on each coarse element. NlMSFEM-M refers to M fine-scale elements for each NlMSFEM basis function.

6.2. Mixed Locality Multiscale Finite Elements. In this section we present preliminary results from the mixed-locality multiscale finite element method. In Figure 6.3 we compare results from simulating a problem of periodic microscale heterogeneity with a horizon, $\delta = 0.1$, large enough to create meaningful difference between the local and nonlocal solutions. It appears, qualitatively, that the ML-MSFEM method reproduces the behavior of the nonlocal solution, despite the coarse global solve being performed with the local model. This is a promising first look at the ML-MSFEM, but needs significant further investigation.

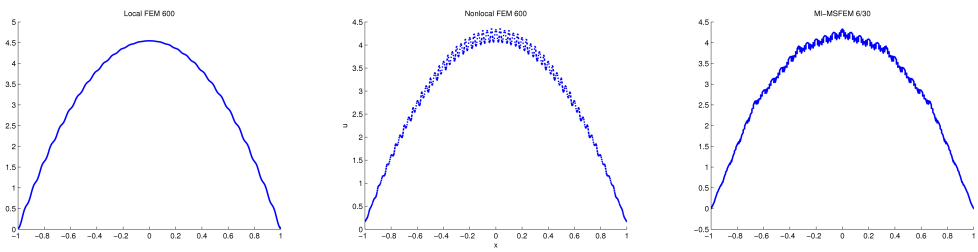


Fig. 6.3: (Left) Local solution with standard FEM, 600 elements. (Middle) Non-local problem solution with standard FEM with 600 elements. (Right) ML-MSFEM solution with 20 coarse elements, 30 fine elements per basis function.

7. Conclusions. In this report we summarized results contained in [4]. We presented preliminary results on a multiscale finite element method for the nonlocal peridynamic theory of continuum mechanics. We presented a novel Galerkin framework intended for a unified analysis of multiscale finite element methods, and demonstrated good performance of the nonlocal and mixed-locality multiscale finite element methods on simple 1D problems that standard finite element methods struggle to solve due to microscale heterogeneity.

The initial results in this report are promising, but much more remains to be done. The analysis of the transition from the Ambulant Galerkin framework to the multiscale finite element framework is encouraging but incomplete. In particular, an error depending on the coarse mesh size H should be discovered in the truncation of the simulation domain for the multiscale basis functions. Also, while the numerical simulations showed that the Nl-MSFEM has H^2 convergence for piece-wise linear microscale basis functions, the analysis in this work demonstrated convergence without an explicit convergence rate. In future work both of these issues will be addressed.

Additionally, analysis of the mixed-locality multiscale finite element method remains to be seen. We anticipate that the asympototic compatibility framework of Tian and Du [26], combined with the horizon limit convergence results for the peridynamic model [25] will pave the way for extending the AFEM framework to allow analysis of the mixed locality multiscale finite element method.

The numerical results in this work were promising, but were restricted to linear problems in 1D. In future work numerical experiments for 2D and 3D problems as well as time dependent and nonlinear problems will be performed. Furthermore, we will implement this method in Sandia's multiphysics finite element code, Albany/LCM [15], in combination with the peridynamics code Peridigm [12]. We anticipate that the methods described here will be particular useful for time-dependent nonlinear materials for which fully implicit time stepping schemes are not an option, as we expect the stability condition to be dependent on the coarse mesh size H.

Ultimately the goal is to communicate, in a mathematically consistent way, from the quantum scale to design-scale models. In this work we presented preliminary results on an important component of that goal: communicating from the lower limits of the scales of the peridynamic theory of continuum mechanics to nonlocal or local models at the design-scale.

REFERENCES

[1] B. Alali and R. Lipton, *Multiscale dynamics of heterogeneous media in the peridynamic formulation*, Journal of Elasticity, 106 (2012), pp. 71–103.

[2] E. Askari, F. Bobaru, R. Lehoucq, M. Parks, S. Silling, and O. Weckner, *Peridynamics for multiscale materials modeling*, in Journal of Physics: Conference Series, vol. 125, IOP Publishing, 2008, p. 012078.

[3] X. Chen and M. Gunzburger, *Continuous and discontinuous finite element methods for a peridynamics model of mechanics*, Computer Methods in Applied Mechanics and Engineering, 200 (2011), pp. 1237–1250.

[4] T. Costa, S. Bond, D. Littlewood, and S. Moore, *Peridynamic multiscale finite element method*, SAND Report, Sandia National Laboratories, Albuquerque, New Mexico, (2015).

[5] Q. Du, M. Gunzburger, R. B. Lehoucq, and K. Zhou, *Analysis and approximation of nonlocal diffusion problems with volume constraints*, SIAM review, 54 (2012), pp. 667–696.

[6] Q. Du, L. Ju, L. Tian, and K. Zhou, *A posteriori error analysis of finite element method for linear nonlocal diffusion and peridynamic models*, Mathematics of Computation, 82 (2013), pp. 1889–1922.

[7] Q. Du, L. Tian, and X. Zhao, *A convergent adaptive finite element algorithm for nonlocal diffusion and peridynamic models*, SIAM Journal on Numerical Analysis, 51 (2013), pp. 1211–1234.

[8] Y. Efendiev and T. Y. Hou, *Multiscale finite element methods: theory and applications*, vol. 4, Springer Science & Business Media, 2009.

[9] U. L. Hetmaniuk and R. B. Lehoucq, *A special finite element method based on component mode synthesis*, ESAIM: Mathematical Modelling and Numerical Analysis, 44 (2010), pp. 401–420.

[10] R. B. Lehoucq and M. P. Sears, *Statistical mechanical foundation of the peridynamic nonlocal continuum theory: Energy and momentum conservation laws*, Physical Review E, 84 (2011), p. 031112.

[11] R. B. Lehoucq and S. A. Silling, *Statistical coarse-graining of molecular dynamics into peridynamics*, Report SAND2007-6410, Sandia National Laboratories, Albuquerque, New Mexico, (2007).

[12] M. L. Parks, D. J. Littlewood, J. A. Mitchell, and S. A. Silling, *Peridigm users guide v1. 0.0*, 2012.

[13] R. Rahman and J. Foster, *Peridynamic theory of solids from the perspective of classical statistical mechanics*, Physica A: Statistical Mechanics and its Applications, (2015).

[14] R. Rahman and A. Haque, *A peridynamics formulation based hierarchical multiscale modeling approach between continuum scale and atomistic scale*, International Journal of Computational Materials Science and Engineering, 1 (2012), p. 1250029.

[15] A. G. Salinger, R. A. Bartett, Q. Chen, X. Gao, G. Hansen, I. Kalashnikova, A. Mota, R. P. Muller, E. Nielsen, J. Ostien, et al., *Albany: A component-based partial differential equation code built on trilinos.*, tech. rep., Sandia National Laboratories Livermore, CA; Sandia National Laboratories (SNL-NM), Albuquerque, NM (United States), 2013.

[16] P. Seleson, M. L. Parks, M. Gunzburger, and R. B. Lehoucq, *Peridynamics as an upscaling of molecular dynamics*, Multiscale Modeling & Simulation, 8 (2009), pp. 204–227.

[17] P. D. Seleson, *Peridynamic multiscale models for the mechanics of materials: constitutive relations, upscaling from atomistic systems, and interface problems*, PhD thesis, Florida State University, 2010.

[18] S. Silling, M. Epton, O. Weckner, J. Xu, and E. Askari, *Peridynamic states and constitutive modeling*, Journal of Elasticity, 88 (2007), pp. 151–184.

[19] S. Silling and R. Lehoucq, *Peridynamic theory of solid mechanics*, Advances in Applied Mechanics, 44 (2010), pp. 73–166.

[20] S. Silling, O. Weckner, E. Askari, and F. Bobaru, *Crack nucleation in a peridynamic solid*, International Journal of Fracture, 162 (2010), pp. 219–227.

[21] S. A. Silling, *Reformulation of elasticity theory for discontinuities and long-range forces*, Journal of the Mechanics and Physics of Solids, 48 (2000), pp. 175–209.

[22] ———, *Linearized theory of peridynamic states*, Journal of Elasticity, 99 (2010), pp. 85–111.

[23] ———, *A coarsening method for linear peridynamics*, International Journal for Multiscale Computational Engineering, 9 (2011).

[24] S. A. Silling and J. V. Cox, *Hierarchical multiscale method development for peridynamics*, tech. rep., Sandia National Laboratories, 2014.

[25] S. A. Silling and R. B. Lehoucq, *Convergence of peridynamics to classical elasticity theory*, Journal of Elasticity, 93 (2008), pp. 13–37.

[26] X. Tian and Q. Du, *Asymptotically compatible schemes and applications to robust discretization of nonlocal models*, SIAM Journal on Numerical Analysis, 52 (2014), pp. 1641–1665.

[27] ———, *Asymptotically compatible schemes and applications to robust discretization of nonlocal models*, SIAM Journal on Numerical Analysis, 52 (2014), pp. 1641–1665.

ROBUSTNESS AND SENSITIVITY OF A FUNCTIONAL TO ESTIMATE THE CONVECTION COEFFICIENT

CARLOS A. GARAVITO-GARZÓN* AND RICHARD B. LEHOUCQ†

Abstract. The purpose of this paper is to investigate an aspect of a PDE constrained optimization approach for estimating the velocity field given image data. Our investigation is accomplished via a set of numerical experiments that show that the functional is robust, sensitive, and largely independent of boundary conditions.

1. Introduction. The purpose of this paper is to investigate an aspect of a PDE constrained optimization approach for estimating the velocity field given image data. In particular, we are interested in understanding the robustness, sensitivity and dependence upon boundary conditions of the functional for the optimization problem (1.1). This is accomplished via a set of numerical experiments. The functional used was analyzed by Ito and Kunisch in [2]; we also consider a minor variation. The functional is the sum of a discrepancy and regularization terms; the former term determines the fit of the model induced by the PDE constraint and the latter term imposes structure upon the velocity field determined. The results of this paper are in support of a PDE constrained optimization approach for digital image correlation; see [3]

The functional is sensitive, roughly, if it increases dramatically when evaluated away from the velocity field of interest, or the target velocity b_*. The functional is robust if we can determine intervals for the regularization parameter β for which the minimum occurs at the target velocity. The numerical experiments presented in sections 3 and 4 will illustrate a robust and sensitive functional.

Let ϕ be the image intensity and $\hat{\phi}$ the corresponding measurements over some domain $\Omega \subset \mathbb{R}^{1,2}$. Given $\hat{\phi}$ we are interested in recovering the velocity field. We investigated the following two objective functions, or functionals,

$$(\phi_*, b_*) = \underset{(\phi,b)\in\mathcal{P}\times\mathcal{B}}{\operatorname{argmin}} \left(\|\phi - \hat{\phi}\|^2_{L^2(\Omega)\times(0,\tau)} + \frac{\beta}{2} \begin{cases} \sigma\,(\tau^2/\hat{b}^2)\|\nabla b\|^2_{L^2(\Omega)} \\ (\tau/\hat{b}^2)\|b\|^2_{L^2(\Omega)} \end{cases} \right) \tag{1.1a}$$

where ϕ is subject to

$$\frac{\partial}{\partial t}\phi + b\cdot\nabla\phi = \sigma\Delta\phi\,, \quad \phi(x,0) = \phi_0(x)\,, \quad x\in\Omega\,, \tag{1.1b}$$

along with homogenous Dirichlet or Neumann boundary conditions. The second functional, containing $\|b\|^2_{L^2(\Omega)}$, was used by Ito and Kunisch in [2] to estimate the velocity b. The norms used for the functional are defined as

$$\begin{aligned} \|\phi - \hat{\phi}\|_{L^2(\Omega)\times(0,\tau)} &\coloneqq \int_\Omega \int_0^\tau \left(\phi - \hat{\phi}\right)^2 dx\,dt\,, \\ \|\nabla b\|^2_{L^2(\Omega)} &\coloneqq \int_\Omega \nabla b : \nabla b\,dx\,, \text{ and} \\ \|b\|^2_{L^2(\Omega)} &\coloneqq \int_\Omega b\cdot b\,dx\,. \end{aligned}$$

*School of Mathematics, University of Minnesota, garav007@umn.edu
†Sandia National Laboratories, rblehou@sandia.gov

We assume without loss of generality that ϕ is a dimensionless quantity, $\mathcal{P}$ and $\mathcal{B}$ are suitable functional spaces, τ and $\hat{b}$ are the time interval and a characteristic velocity associated with the geometry of the problem, respectively. The parameter σ represents the diffusion coefficient and b_* and ϕ_* are the solutions to our inverse problem. From an experimental perspective, b and σ represent the mean and variance of the measurements. The purpose of the scaling parameters $\sigma\,(\tau^2/\hat{b}^2)$ and $\tau/\hat{b}^2$ are to ensure that both terms in the functional have the same units. This enables the effect of the regularization parameter β to be ascertained in the numerical experiments.

The velocity solution to the optimization problem (1.1) is not unique; any vector field $\bar{b}$ orthogonal to $\nabla\hat{\phi}$ grants that $b_* + \bar{b}$ is also a minimizer. Hence without loss of generality, we suppose that $\nabla \cdot b_* = 0$. This constraint renders a well-posed optimization problem; see Ito and Kunisch in [2] for further details.

We determine the robustness and sensitivity through a set of numerical experiments. The numerical experiments are driven by exploiting the relationship between the PDE constraint (1.1b) and the diffusion process given by the Itô stochastic differential equation (SDE). This relationship is the subject of Section 2. Our numerical experiments are summarized in sections 3 and 4 for one and two-dimensions, respectively. Concluding remarks are provided in section 5. Our findings suggest that the use of mathematically sophisticated techniques for the recovery of b from experimental data, e.g., PDE constrained optimization is a viable approach.

2. Stochastic Model. The first term in the objective function (1.1) needs to discern the difference between measured intensity values and fit with the diffusion equation. Suppose that $\Omega = \mathbb{R}^1$ so that PDE constraint (1.1b) simplifies to

$$\begin{aligned} \frac{\partial \phi}{\partial t} + b\frac{\partial \phi}{\partial x} &= \sigma\,\frac{\partial^2 \phi}{\partial x^2} \qquad 0 < t \le \tau\,, \text{ over } \mathbb{R}\,, \\ \phi(x,0) &= \phi_0(x)\,, \end{aligned}$$

where ϕ_0 is nonnegative and $\int_{\mathbb{R}} \phi_0(x)\,dx = 1$, i.e., ϕ_0 is a probability density on the real line. The solution ϕ is then given by

$$\phi(x,t) = \frac{1}{\sqrt{4\pi\sigma\,t}} \int_{\mathbb{R}} e^{-(x-y-bt)^2/(4\sigma\,t)} \phi_0(y)\,dy\,.$$

This expression is understood as follows: $\phi_0(y)\,dy$ is the probability over a region dy about y. The product $e^{-(x-y-bt)^2/(4\sigma\,t)}\phi_0(y)\,dy$ scales this probability by its distance from x at time t. Normalizing the scaling and summing over all y results in an average, or expectation $\phi(x,t)$.

Let $\mathbb{E}\big[X_t \,|\, X_0 = x\big]$ denote the expectation of the random variable X_t conditioned upon $X_0 = x$. The solution of the diffusion equation also has the well-known probabilistic identification

$$\phi(x,t) = \mathbb{E}^x\big[\phi_0(X_t)\big] = \mathbb{E}\big[\phi_0(X_t) \,|\, X_0 = x\big]\,,$$

where X_t is a random variable satisfying the stochastic differential equation (SDE)

$$dX_t = b\,dt + \sqrt{2\sigma}\,dW_t\,, \quad X_0 = x\,, \tag{2.1}$$

where W_t is a Wiener process. This interpretation has the benefit of providing a mechanism to simulate the process associated with the diffusion equation. The solution $\phi(x,t)$ is the expectation of the random variable X_t conditioned on $X_0 = x$; this expectation can be computed either

- by evaluating the integral (via quadrature) or
- estimating via a Monte Carlo simulation using the SDE; a collection of realizations is averaged to provide an approximation. This means

$$\mathbb{E}^x\big[\phi_0(X_t)\big] \approx \frac{1}{N}\sum_{i=1}^{N}\phi_0(X_t^i), \tag{2.2}$$

where N denotes the number of realizations or microstates.

We make the modeling assumption that the measured image intensity is exact but that the underlying trajectory evolves according to the SDE. The difference,

$$\phi(x,t) - \hat{\phi}(x,t) = \mathbb{E}^x\big[\phi_0(X_t)\big] - \phi_0(X_t)\,,$$

then represents the fluctuation about the mean of the intensity of a trajectory. The quantity $\phi_0(X_t)$ is determined via an application of the stochastic chain rule, i.e., Itô's Lemma to obtain

$$d\phi_0(X_t) = \left(b\,\frac{d\phi_0(X_t)}{dX_t} + \sigma\frac{d^2\phi_0(X_t)}{dX_t^2}\right)dt + \sqrt{2\sigma}\,\frac{d\phi_0(X_t)}{dX_t}dW_t\,. \tag{2.3}$$

We use the Euler-Maruyama method to integrate the SDE (2.1); see, e.g., [1]. We estimate the value of $\phi(x,t) = \mathbb{E}^x\big[\phi_0(X_t)\big]$ using the Monte Carlo method presented in (2.2).

We estimate the values of b and σ by Monte Carlo approximations for the mean $\mathbb{E}^x[X_t]$ and variance

$$\mathbb{V}^x[X_t] := \mathbb{E}^x[X_t^2 - \big(\mathbb{E}^x[X_t]\big)^2]\,.$$

The estimates can be compared with the closed-form expressions

$$\mathbb{E}^x[X_t] = x + b\,t\,,\ \mathbb{V}^x[X_t] = 2\sigma\,t\,, \tag{2.4}$$

for the mean and variance of (2.1). The expressions explain that the centered mean is proportional to time, i.e.,

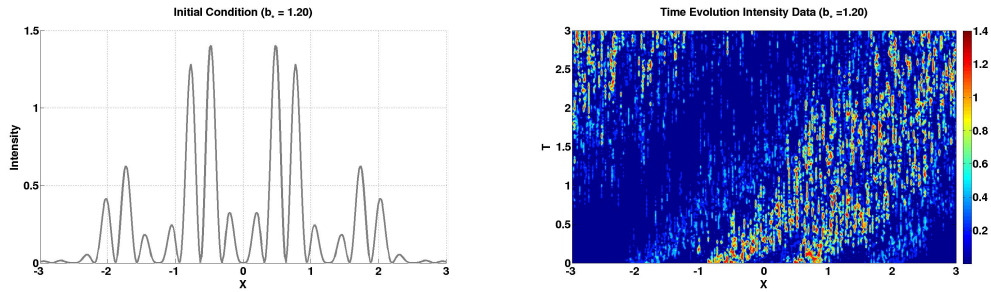
$$\mathbb{E}^x[X_t] - x \propto t\,,$$

with constant b—the velocity. The standard deviation $\sqrt{\mathbb{V}^x[X_t]}$ provides the standard error for the centered mean. Comparing the estimates of the mean and variance serve as a check on both our understanding and implementation of the SDE. However, the main interest is in determining the effect of boundary conditions, i.e., consideration of a finite domain Ω and the role of either homogenous Dirichlet or Neumann boundary conditions, upon the robustness and sensitivity of the functional. The generalization of these results to two dimensions is straightforward and will not be presented.

3. One-Dimensional Experiments.

3.1. Robustness and Sensitivity. The numerical experiments presented in this section will show that the functional (1.1a) is both and robust and sensitive for several types of boundary conditions.

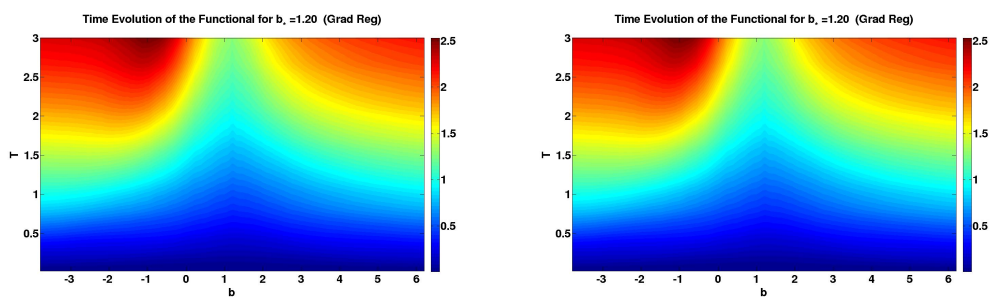
We consider the initial condition $\phi_0(x) = (\sin(2x) + \sin(3x))^2\sin^2(10x)/\gamma$ where γ is a normalization constant. We set $\Omega = (-3,3)$ and $b_* = 1.2$. For the computation of the integrals we use a trapezoidal rule in both space and time. The value of $\hat{\phi}$ is computed using Itô's Lemma (2.3). The initial condition and its time evolution are displayed in Figure 3.1.



(a) Initial Condition.

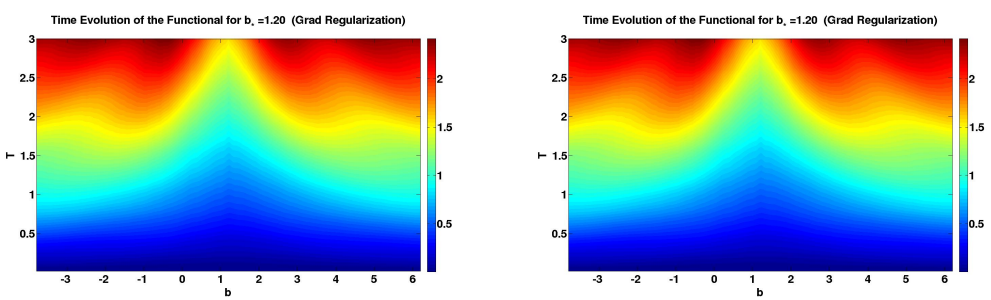
(b) Time evolution of $\hat{\phi}$ for periodic boundary condition.

Fig. 3.1: Initial condition (left) and time evolution of $\hat{\phi}$ (right) using periodic boundary condition.



(a) $10^{-3}\|\nabla b\|^2_{L^2(\Omega)}$ regularization and Neumann boundary condition.

(b) $\|b\|^2_{L^2(\Omega)}$ regularization and Neumann boundary condition.

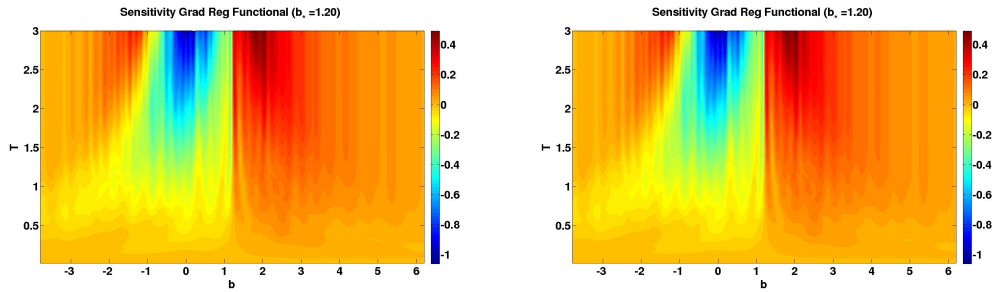


(c) $10^{-3}\|\nabla b\|^2_{L^2(\Omega)}$ regularization and no boundary condition.

(d) $\|b\|^2_{L^2(\Omega)}$ regularization and no boundary condition.

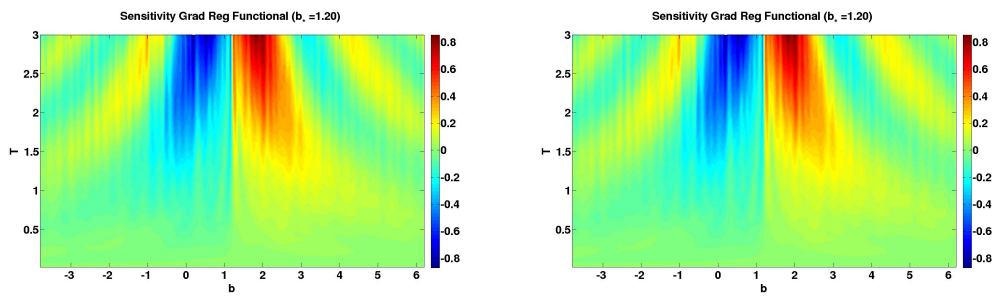
Fig. 3.2: Functional with gradient(lefmost) and $10^{-3}\|b\|^2_{L^2(\Omega)}$(rightmost) regularizations with Neumann(top) and no (bottom) boundary conditions for $b_* = 1.2$, $\sigma = 0.1$.

Figure 3.2 plots the evolution of the functional over $\Omega = (-3, 3)$ for both functionals (1.1a) where $\beta = 10^{-3}$. Figure 3.3 displays the corresponding sensitivity plots. The sensitivity is approximated numerically using central finite differences with respect to b. For both figures, the time slices increase in variation away from the target velocity with increas-



(a) $10^{-3}\|\nabla b\|^2_{L^2(\Omega)}$ Regularization and Neumann boundary condition.

(b) $\|b\|^2_{L^2(\Omega)}$ Regularization and Neumann boundary condition.



(c) $10^{-3}\|\nabla b\|^2_{L^2(\Omega)}$ Regularization and no boundary condition.

(d) $\|b\|^2_{L^2(\Omega)}$ Regularization and no boundary condition.

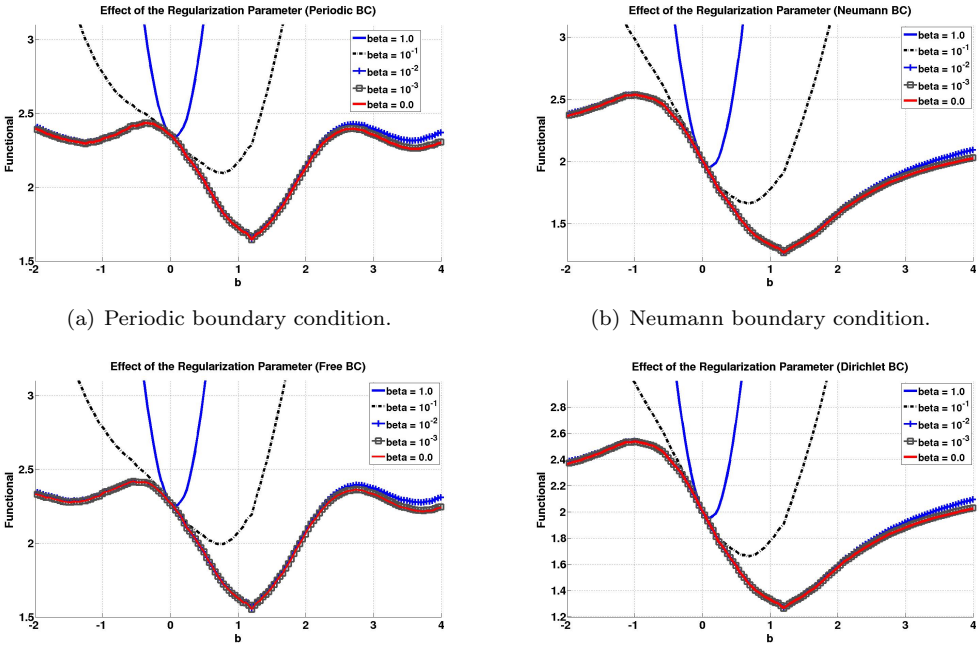
Fig. 3.3: Sensitivity of the functional with gradient(leftmost) and $10^{-3}\|b\|^2_{L^2(\Omega)}$(rightmost) regularizations with Neumann(top) and no(bottom) boundary conditions for $b_* = 1.2$, $\sigma = 0.1$.

ing time. This indicates that we can recover the value of b_* when the measurements $\hat{\phi}$ are performed for a sufficient amount of time. Due to space constraints we only report the data corresponding to Neumann and the no boundary conditions.

The robustness of the functional is displayed in Figure 3.4. The plots demonstrate the effect of the regularization parameter β for several types of boundary conditions. As β is decreased, the minimum value of the functional is reached at $b_* = 1.2$. This behavior is compatible with the definition of *robustness* given at the start of Section 1 and explains that if β is too large, then the functional renders an inaccurate minimum. The experiments suggest that both the sensitivity and robustness of the functionals (1.1a) are independent of the boundary conditions.

3.2. Effect of Boundary Conditions. In order to characterize the effect of boundary conditions upon the estimate of the velocity the stochastic process, we consider the domain $\Omega = (-30.0, 30.0)$. We condition on a uniform partition of the ball $B_{10^{-2}}(0)$ consisting of 900 points, the time-step for the temporal integration is $\Delta t = 10^{-4}$, $\sigma = 10^{-1}$ and we evolve the system for 20 seconds. We use the random number generator provided by Matlab *randn* for the simulation of the Wiener process W_t. We approximate simulating over the real line by imposing periodic boundary conditions on the system.

Figure 3.5 compares estimates of the mean and variance with their closed form expressions for the velocity fields $b_* = -2, \sim \mathcal{N}(-2.0, .1)$ where $\mathcal{N}(\mu, \sigma)$ is the normal distribution



(a) Periodic boundary condition.

(b) Neumann boundary condition.

(c) No boundary condition.

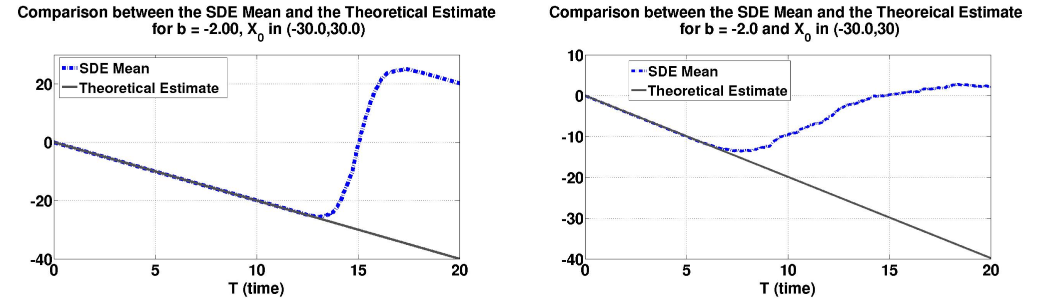
(d) Dirichlet boundary condition.

Fig. 3.4: Robustness of the functional for several values of the regularization parameter β using periodic (top-left), Neumann (top-right), no (bottom-left), and Dirichlet(bottom-right) boundary conditions at time $t = 3.0$.

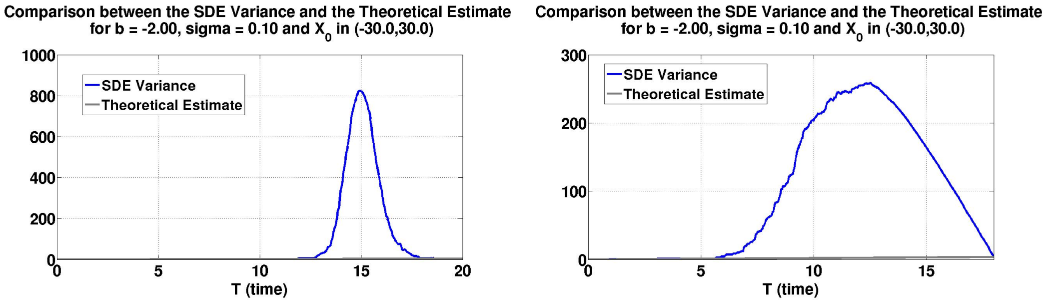
with mean μ and variance σ. The plots demonstrate that the estimates derived from the data are accurate until approximately $t = 13$ and $t = 5$ for $b_* = -2.0$ and $\mathcal{N}(-2.0, .1)$, respectively. The explanation is that because of the periodic boundary condition, when the diffusion hits one of the boundary points, the diffusion is moved near the other boundary point introducing a jump for the diffusion. The estimates no longer correspond to the expressions(2.4) derived under the assumption that the SDE is driven by a Wiener process.

We provide a general characterization of the effect of the boundary condition in Figure 3.6 where, we display the $\log_{10}$ error for the mean (leftmost) and variance (rightmost) using periodic (green), Dirichlet (red), Neumann (black), and no (blue) boundary conditions. Our results show that in the periodic case, the error is the biggest one for both mean and variance. In the variance case the error due periodic boundary conditions is several orders of magnitude bigger than for the Dirichlet and Neumann types.

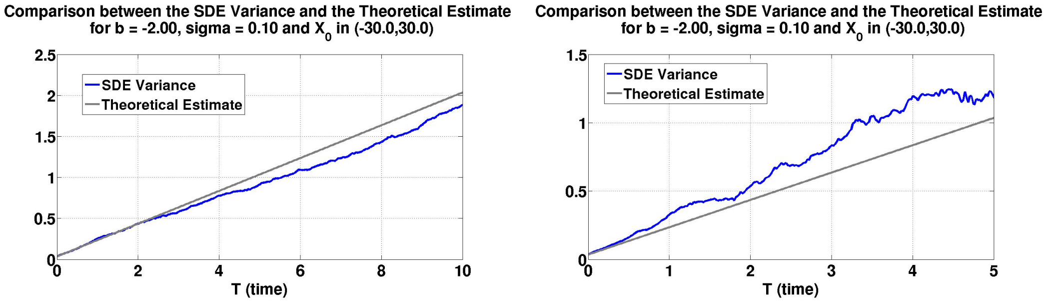
We conclude from both Figures 3.5–3.6 that the imposition of boundary conditions has no effect until the diffusion approaches the boundary. Hence if x is near the center of the finite domain Ω and the time interval is not large, then the boundary conditions are irrelevant.



(a) $b_* = -2$ (b) $b_* \sim \mathcal{N}(-2.0, .1)$

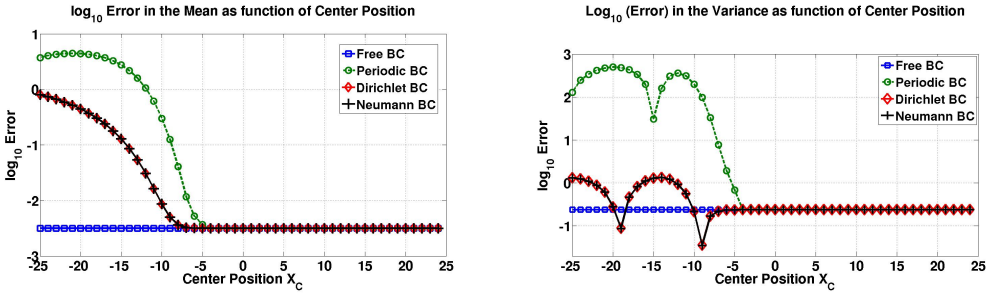


(c) $b_* = -2$ (d) $b_* \sim \mathcal{N}(-2.0, .1)$



(e) $b_* = -2$ (f) $b_* \sim \mathcal{N}(-2.0, .1)$

Fig. 3.5: Mean and variance for constant velocity (leftmost) and a spatial varying velocity with mean -2 (rightmost) with $\sigma = 0.1$. The estimates deviate from the closed form expressions due to the imposition of periodic boundary conditions.



(a) Logarithm of the error for the SDE Mean

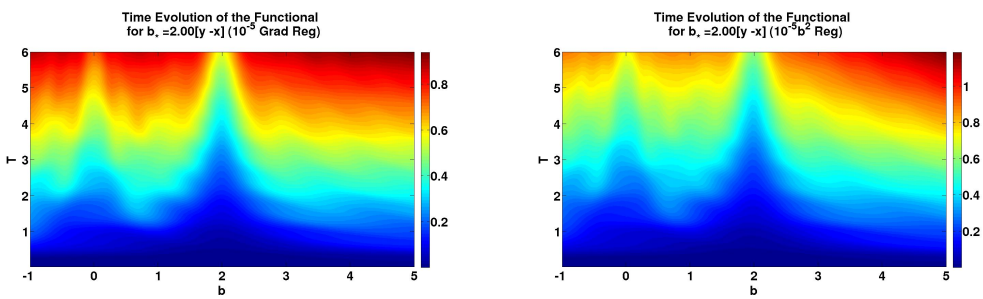
(b) Logarithm of the error for the SDE Variance

Fig. 3.6: Logarithm error for the mean (leftmost) and variance (rightmost) as function of the initial distribution center for a target velocity $b_* = -1.0$ and $\sigma = 0.1$ using for types of boundary conditions periodic (green), Dirichlet (red), Nuemann (black) and the free space solution (blue).

4. Two-Dimensional Experiments. We consider cases where the velocity has the following form $\mathbf{b}_* = 2\,(y, -x)$ so that $b_* = 2$. We compare $\|\nabla b\|^2_{L^2(\Omega)}$ and $\|b\|^2_{L^2(\Omega)}$ regularizations. The initial condition is a Gaussian pulse $\phi_0(x, y) = \frac{1}{\pi}\exp\left(-x^2 - y^2\right)$. The SDE integration is performed over the time interval $(0, 6)$ with time step is $\Delta t = 10^{-4}$. We conclude that these two regularizations have an equivalent behavior on the sensitivity of the functional. This fact is important for optimization techniques such as PDE constrained optimization, where the treatment of the $\|b\|^2_{L^2(\Omega)}$ is less difficult than its gradient counterpart.

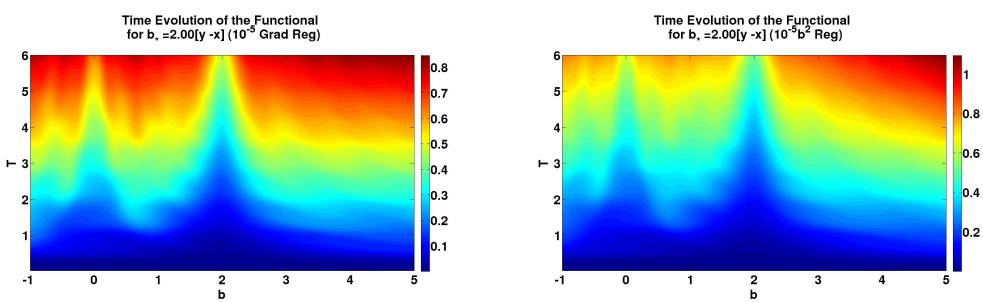
Figure 4.1 displays the functionals for increasing time and the velocity parameter b. The data indicates that the functional has a global minimum at $b = b_* = 2$. In Figure 4.1, we display only the results corresponding to Neumann (top) and free (boundary) conditions.

The sensitivity of the functional is displayed in Figure 4.2 for Neumann (top) and no (bottom) boundary conditions. This Figure shows that the functional is indeed sensitive provided that the time $t > 1$. Notice that, the time to attain a prescribed level sensitivity is larger for than for a one-dimensional problem. Figure 4.1 and 4.2 show the existence of a local minimum at $b = 0$. Nevertheless, this local minimum is less sensitive than the one at $b = b_* = 2$ (Figure 4.2) since the region where the derivative changes the most is larger when $b = b_* = 2$.



(a) $10^{-5}\|\nabla b\|^2_{L^2(\Omega)}$ Regularization and Neumann boundary condition.

(b) $\|b\|^2_{L^2(\Omega)}$ Regularization and Neumann boundary condition.

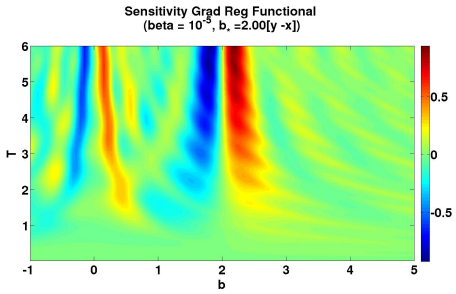


(c) $10^{-5}\|\nabla b\|^2_{L^2(\Omega)}$ Regularization and no boundary condition.

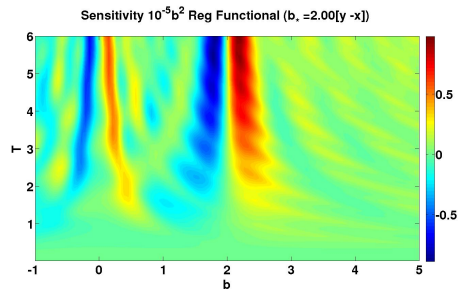
(d) $\|b\|^2_{L^2(\Omega)}$ Regularization and no boundary condition.

Fig. 4.1: Functional in 2D with $10^{-5}\|\nabla b\|^2_{L^2(\Omega)}$ (leftmost) and $10^{-5}\|b\|^2_{L^2(\Omega)}$(rightmost) regularizations with Neumann(top) and no(bottom) boundary conditions for $\mathbf{b}_* = 2.0\,(y, -x)$ and $\sigma = 0.1$.

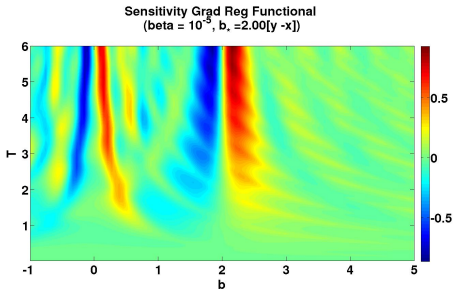
The robustness of the functional is displayed in Figure 4.3 for the $\|\nabla b\|^2_{L^2(\Omega)}$ (leftmost)



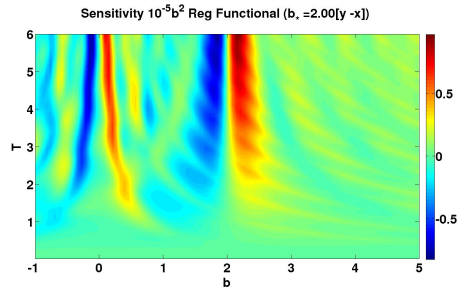
(a) $10^{-5}\|\nabla b\|^2_{L^2(\Omega)}$ Regularization and Neumann boundary condition.



(b) $\|b\|^2_{L^2(\Omega)}$ Regularization and Neumann boundary condition.



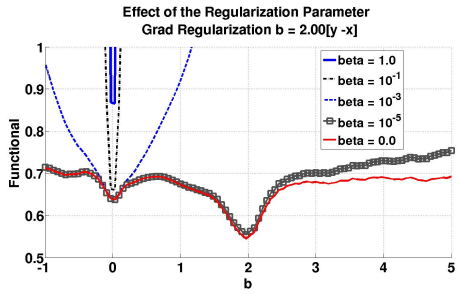
(c) $10^{-5}\|\nabla b\|^2_{L^2(\Omega)}$ Regularization and no boundary condition.



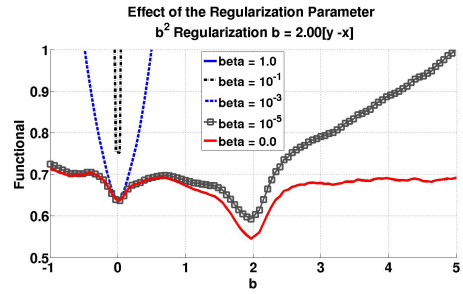
(d) $\|b^2\|^2_{L^2(\Omega)}$ Regularization and no boundary condition.

Fig. 4.2: Sensitivity of the functional in 2D with $10^{-5}\|\nabla b\|^2_{L^2(\Omega)}$ (leftmost) and $10^{-5}\|b\|^2_{L^2(\Omega)}$(rightmost) regularizations with Neumann (top) and (bottom) no boundary conditions for $\mathbf{b}_* = 2.0\,(y, -x)$ and $\sigma = 0.1$.

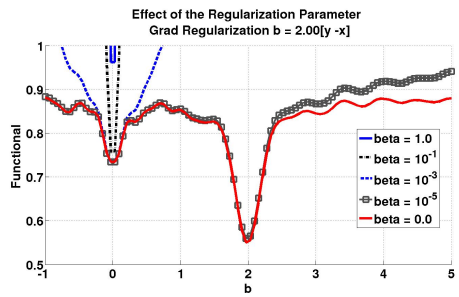
and $\|b\|^2_{L^2(\Omega)}$ regularizations, respectively. The behavior of these types of regularizations are comparable; for $\beta = 10^{-5}$ both approaches have a global minimum at $b_* = 2$. However, the $\|b\|^2_{L^2(\Omega)}$ regularization varies away from the minimum.



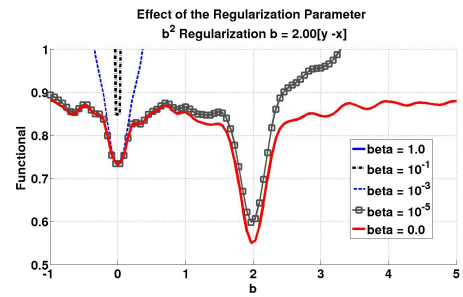
(a) Periodic boundary condition and $\|\nabla b\|^2_{L^2(\Omega)}$ regularization.



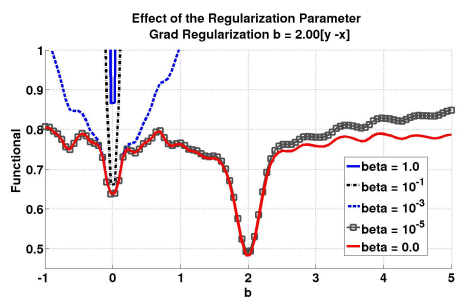
(b) Periodic boundary condition and $\|b\|^2_{L^2(\Omega)}$ regularization.



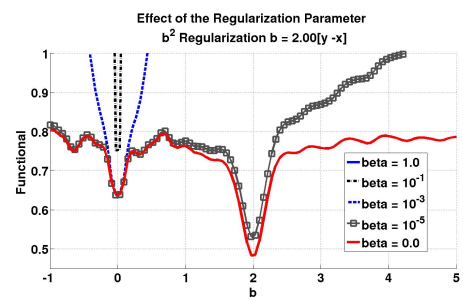
(c) Neumann boundary condition and $\|\nabla b\|^2_{L^2(\Omega)}$ regularization.



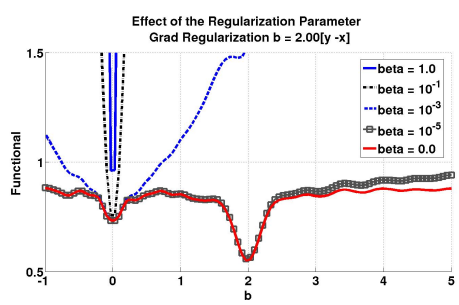
(d) Neumann boundary condition and $\|b\|^2_{L^2(\Omega)}$ regularization.



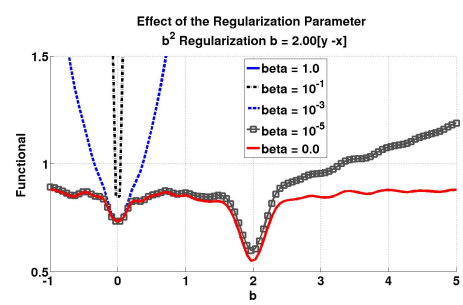
(e) No boundary condition and $\|\nabla b\|^2_{L^2(\Omega)}$ regularization.



(f) No boundary condition and $\|b\|^2_{L^2(\Omega)}$ regularization.



(g) Dirichlet boundary condition and $\|\nabla b\|^2_{L^2(\Omega)}$ regularization .



(h) Dirichlet boundary condition and $\|b\|^2_{L^2(\Omega)}$ regularization.

Fig. 4.3: Robustness of the functional in two-dimensions for $\|\nabla b\|^2_{L^2(\Omega)}$ (left) and $\|b\|^2_{L^2(\Omega)}$ regularizations using a range of values of the parameter β using (from top to bottom) periodic, Neumann, no, and Dirichlet boundary conditions at time $t = 6.0$.

5. Conclusions. We presented numerical experiments in one- and two- dimensions suggesting that both functionals (1.1) are robust and sensitive given a sufficient time τ. A functional is sensitive, roughly, if it increases dramatically when evaluated away from the velocity field of interest, or the target velocity b_*. The functional is robust if we can determine intervals for the regularization parameter β for which the minimum of the functional occurs for the target velocity. Moreover, our conclusion is also independent of the imposition of the Dirichlet or Neumann boundary conditions. This is in contrast to a statistical estimation that renders inaccurate estimates unless τ is not large enough so that the diffusion hits the boundary.

REFERENCES

[1] D. J. Higham, *An algorithmic introduction to numerical simulation of stochastic differential equations*, SIAM review, 43 (2001), pp. 525–546.

[2] K. Ito and K. Kunisch, *Estimation of the convection coefficient in elliptic equations*, Inverse problems, 13 (1997), p. 995.

[3] R. B. Lehoucq, D. Z. Turner, and C. A. Garavito-Garzón, *PDE constrained optimization for digital image correlation*, Technical Report SAND2015-8515, Sandia National Laboratories, 2015.

ACTIVE SUBSPACES FOR CFD/MHD AND ESTIMATING THE EFFECT OF DIMENSION TRUNCATION ON PROBABILISTIC QUANTITIES OF INTEREST

ANDREW T. GLAWS*, TIMOTHY M. WILDEY†, AND JOHN N. SHADID‡

Abstract. In this paper, we give a concise overview of an emerging dimension reduction approach known as active subspaces. We describe the process of identifying an active subspace as well as an approach for building response surfaces on the lower-dimensional space and utilizing the response surface to estimate the probability of events. The methods are applied to standard verification problems in computational fluid dynamics (CFD) and magnetohydrodynamics (MHD) with known analytic solutions in order to relate the structure of the active variables to the physics in the problems. We also apply the approach to a more challenging CFD problem involving coupled heat transfer and fully-developed turbulent flow in a 3-dimensional pipe using a Spallart-Allmaras Reynolds Averaged Navier Stokes (SARANS) model discretized using stabilized finite elements. In addition, we compare optimization and statistical approaches for estimating the dimension truncation error and show how these estimates can be used to compute upper and lower bounds on probabilistic quantities of interest, such as the probability of a quantity of interest exceeding a given threshold.

1. Introduction. Uncertainty and error are ubiquitous in predictive modeling and simulation due to unknown model parameters and various sources of deterministic and stochastic error. Consequently, there is considerable interest in developing efficient and accurate methods to quantify the uncertainty in the outputs of a computational model. For many models, a scalar quantity of interest is sufficient to describe the results. This may be an average value over the domain or the value at some particular point of interest. Monte Carlo techniques are the standard approach to quantify the uncertainty in a quantity of interest from a computational model. The general appeal of these methods can be attributed to their relative ease of implementation and the fact that they effectively circumvent the *curse of dimensionality*. Unfortunately, the number of samples required to accurately estimate certain probabilistic quantities, especially the probability of rare events, may be prohibitively large for models that require even modest computational costs. Improvements such as importance sampling can greatly reduce the computational cost, but the number of model evaluations often remains prohibitively large.

Alternative approaches based on response surface approximations have grown in popularity in recent years as a means to alleviate the computational burden of brute-force Monte Carlo for computationally intensive models. A wide variety of techniques have been developed, e.g., piecewise-linear approximations requiring derivative information [21, 20, 12, 2], global polynomial approximations computed using stochastic spectral methods [15, 26, 25], and tensor product and sparse grid collocation methods [1, 13, 14, 4, 19]. Unfortunately, all of these approaches suffer from the curse of dimensionality (some more than others) and their application becomes intractable beyond a certain number of parameters. Consequently, there is considerable interest in developing mathematical approaches for dimension reduction.

Common techniques for reducing the dimension involve eliminating parameters which

*Colorado School of Mines, aglaws@mines.edu,

†Optimization and Uncertainty Quantification Department, Center for Computing Research, Sandia National Laboratories, Albuquerque, NM 87185, tmwilde@sandia.gov. Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.,

‡Computational Mathematics Department, Center for Computing Research, Sandia National Laboratories, jnshadi@sandia.gov

appear to have little influence on the quantity of interest. The concept of active subspaces generalizes this approach by searching for directions (subspaces) within the parameter space which capture the majority of the variation in the response as a function of the "active parameters" which are linear combinations of the original parameters. This approach has a solid mathematical foundation [9] and has been very successful in a range of applications [10, 11, 18].

The goal of this paper is to investigate the application of active subspace techniques on both simple and challenging problems in CFD and MHD. We are also interested in understanding the error in probabilistic quantities of interest, e.g., the probability of a quantity of interest exceeding a threshold value, due to the dimension truncation error. We explore both statistical estimates for the dimension truncation error as well as an optimization-based approach.

The remainder of this paper is organized as follows. In Section 2, we describe the fundamental concepts associated with active subspaces and provide algorithms for computing the active subspace and for building a response surface approximation in the active subspace. Section 3 describes how the response surface approximation in the active subspace can be used to estimate the probability of events and how we can use estimates of the dimension truncation error to provide upper and lower bounds on the estimate of the probability. Numerical results are presented in Section 4 and our concluding remarks are given in Section 5.

2. Active Subspaces. We assume that we are given a computational model and that a useful scalar quantity of interest can be obtained from this model. The quantity of interest is written $f(\boldsymbol{x})$ where the vector $\boldsymbol{x} \in \Omega \subset \mathbb{R}^m$ is the collection of model input parameters. The vector length m is referred to as the dimension of the problem. Furthermore, we assume we are given a probability density function $\rho(\boldsymbol{x})$ on the input parameters. Finally, we assume that the quantity of interest depends sufficiently smoothly on the input parameters so that we can define and approximate the gradient of the quantity of interest with respect to the model parameters, $\nabla_{\boldsymbol{x}} f(\boldsymbol{x})$. For the examples in this paper, gradients are approximated using an adjoint-based approach [21, 17, 8]. There are several techniques that can be used if the gradients are unavailable, such as approximating the gradients using finite differences or using surrogate approximations, but this is beyond the scope of this paper.

2.1. Computing the Active Subspace. The computation of the active subspaces begins with the construction of the symmetric, positive semidefinite, $m \times m$ matrix

$$C = \int_{\Omega} \left(\nabla_{\boldsymbol{x}} f(\boldsymbol{x})\right) \left(\nabla_{\boldsymbol{x}} f(\boldsymbol{x})\right)^T \rho(\boldsymbol{x})\, d\boldsymbol{x}. \tag{2.1}$$

Since C is symmetric, it has a real eigenvalue decomposition

$$C = W \Lambda W^T \tag{2.2}$$

where

$$\Lambda = \begin{bmatrix} \lambda_1 & & \\ & \ddots & \\ & & \lambda_m \end{bmatrix} \qquad \text{with} \qquad \lambda_1 \geq \cdots \geq \lambda_m \geq 0 \tag{2.3}$$

and

$$W = \begin{bmatrix} | & & | \\ \boldsymbol{w}_1 & \dots & \boldsymbol{w}_m \\ | & & | \end{bmatrix} \qquad \text{with} \qquad \boldsymbol{w}_i^T \boldsymbol{w}_j = \delta_{ij},\ 1 \leq i, j \leq m. \tag{2.4}$$

The key property of the eigenvalues and eigenvectors of the C matrix is the manner in which they capture the behavior of $f(\boldsymbol{x})$,

$$\int_{\Omega} \left((\nabla_{\boldsymbol{x}} f(\boldsymbol{x}))^T \boldsymbol{w}_i \right)^2 \rho(\boldsymbol{x})\, d\boldsymbol{x} = \lambda_i, \qquad 1 \leq i \leq m. \tag{2.5}$$

The value $(\nabla_{\boldsymbol{x}} f(\boldsymbol{x}))^T \boldsymbol{w}_i$ is precisely the directional derivative of $f(\boldsymbol{x})$ in the direction of $\boldsymbol{w}_i$. Hence, Equation 2.5 says that the average gradient value of $f(\boldsymbol{x})$ in the direction $\boldsymbol{w}_i$ is λ_i. That is, for large values of λ_i, we can expect that the quantity of interest will vary greatly along $\boldsymbol{w}_i$, and for small values of λ_i, the quantity of interest will vary little along $\boldsymbol{w}_i$.

Ideally, the active subspace would be determined by the selection of $1 \leq n < m$ such that the sum of the remaining eigenvalues would be relatively small which would limit the dimension truncation error. However, it is often more beneficial to select the active subspace so that there is a large gap between λ_n and λ_{n+1}. It is often easiest in practice to approximate the integral in Equation (2.1) through Monte Carlo sampling, and selecting n as described will minimize the error in the approximated active subspace [9]. Once an appropriate value of n is selected, the eigenvalue and eigenvector matrices are subdivided into

$$\Lambda = \begin{bmatrix} \Lambda_1 & \\ & \Lambda_2 \end{bmatrix} \quad \text{and} \quad W = \begin{bmatrix} W_1 & W_2 \end{bmatrix} \tag{2.6}$$

where $\Lambda_1 \in \mathbb{R}^{n \times n}$, $\Lambda_2 \in \mathbb{R}^{(m-n)\times(m-n)}$, $W_1 \in \mathbb{R}^{m \times n}$, and $W_2 \in \mathbb{R}^{m \times (m-n)}$. That is, Λ_1 contains the first n eigenvalues and W_1 contains the first n eigenvectors. This subdivision defines the basis for the active and inactive subspaces. Furthermore, it allows for the definition of the so called active and inactive variables

$$\boldsymbol{y} = W_1^T \boldsymbol{x} \qquad \text{and} \qquad \boldsymbol{z} = W_2^T \boldsymbol{x}, \tag{2.7}$$

respectively. The true (physical) input parameters is then given by

$$\boldsymbol{x} = W_1 \boldsymbol{y} + W_2 \boldsymbol{z}. \tag{2.8}$$

Equation 2.8 has broken the original m-dimensional input parameter into an n-dimensional active variable and an $(m-n)$-dimensional inactive variable. From Equation 2.5, it is clear that variations in the active variable $\boldsymbol{y}$ will, on average, have significantly greater impacts on the quantity of interest than variations in the inactive variable $\boldsymbol{z}$.

For high dimension problems, computation of the C matrix given by Equation 2.1 is difficult. However, it can be easily approximated using Monte Carlo integration. Furthermore, the approximation is likely to result in k accurate eigenvalues and eigenvectors if $\alpha k \log(m)$ samples are used. The parameter α is an oversampling factor that is generally chosen to be between 2 and 10. Thus, a practical algorithm for computation of the active subspace is given in Algorithm 1.

2.2. Building Response Surface Approximations on the Active Subspace. Using Equation 2.8, the quantity of interest can be rewritten in terms of the active and inactive variables

$$f(\boldsymbol{x}) = f\left(W_1 \boldsymbol{y} + W_2 \boldsymbol{z}\right). \tag{2.9}$$

The goal is to find a function of the active variable which accurately approximates the full function, $g(\boldsymbol{y}) \approx f\left(W_1^T \boldsymbol{y} + W_2^T \boldsymbol{z}\right)$. One natural choice is to average over the inactive variable for any value of the active variable. That is,

$$g(\boldsymbol{y}) = \int f\left(W_1 \boldsymbol{y} + W_2 \boldsymbol{z}\right) \rho_{\boldsymbol{z}|\boldsymbol{y}}(\boldsymbol{z})\, d\boldsymbol{z} \tag{2.10}$$

Algorithm 1 Computation of the Active Subspace

1: Draw $N = \alpha k \log(m)$ random samples of the parameter space according to $\rho(\boldsymbol{x})$
2: For each $\boldsymbol{x}_i$, calculate the quantity of interest $f(\boldsymbol{x}_i)$ and the gradient $\nabla_{\boldsymbol{x}} f(\boldsymbol{x}_i)$
3: Approximate the matrix C by

$$C \approx \frac{1}{N} \sum_{i=1}^{N} \left(\nabla_{\boldsymbol{x}} f(\boldsymbol{x}_i)\right) \left(\nabla_{\boldsymbol{x}} f(\boldsymbol{x}_i)\right)^T$$

4: Take the eigenvalue decomposition $C = W \Lambda W^T$
5: Select $1 \leq n < m$ and define

$$\Lambda = \begin{bmatrix} \Lambda_1 & \\ & \Lambda_2 \end{bmatrix}, \quad W = \begin{bmatrix} W_1 & W_2 \end{bmatrix}$$

where Λ_1 contains the first n eigenvalues and W_1 contains the first n eigenvectors

where $\rho_{\boldsymbol{z}|\boldsymbol{y}}$ is the conditional probability of $\boldsymbol{z}$ given $\boldsymbol{y}$. Using this response surface, we get that

$$\int \left(f(\boldsymbol{x}) - g(W_1^T \boldsymbol{x})\right)^2 \rho(\boldsymbol{x})\, d\boldsymbol{x} \leq C \left(\lambda_{n+1} + \cdots + \lambda_m\right) \tag{2.11}$$

where C depends on the density function $\rho(\boldsymbol{x})$. Thus, Equation (2.10) provides a low-dimensional response surface whose root-mean-squared error is on the order of the sum of the last $m - n$ eigenvalues. That is, errors in $g(\boldsymbol{y})$ are contained by the contribution of the inactive variable to $f\left(W_1 \boldsymbol{y} + W_2 \boldsymbol{z}\right)$.

Algorithm 2 Building a Response Surface in the Active Subspace

1: Draw N samples from the active subspace according to the marginal density

$$\pi_{\boldsymbol{y}}(\boldsymbol{y}) = \int \rho\left(W_1 \boldsymbol{y} + W_2 \boldsymbol{z}\right)\, d\boldsymbol{z}$$

2: For each $\boldsymbol{y}_i$, draw M samples from the inactive subspace according to the conditional density $\pi_{\boldsymbol{z}|\boldsymbol{y}_i}(\boldsymbol{z})$
3: For each $\boldsymbol{x}_{ij} = W_1 \boldsymbol{y}_i + W_2 \boldsymbol{z}_j$, calculate the quantity of interest $f(\boldsymbol{x}_{ij})$
4: Approximate the average value of each point in the active subspace using

$$\overline{g}(\boldsymbol{y}_i) \approx \frac{1}{N} \sum_{j=1}^{M} f(\boldsymbol{x}_{ij})$$

5: Construct a response surface $g(\boldsymbol{y})$ using the N points $(\boldsymbol{y}_i, \overline{g}(\boldsymbol{y}_i))$

Similar to before, this function is difficult to compute for all $\boldsymbol{y}$. Hence, we choose several values of the active variable and approximate Equation 2.10 using Monte Carlo integration at these points. Once this is done a surrogate can be constructed on the low-dimensional space using any of the well-developed techniques available. A practical process for constructing $g(\boldsymbol{y})$ is given in Algorithm 2.

3. Estimating Probabilities using the Response Surface in the Active Subspace. In many applications, estimating the probability of a certain event has great practical importance. Given the quantity of interest from a model and the probability distribution $\rho(\boldsymbol{x})$ on the inputs, one might ask for the probability that $f(\boldsymbol{x})$ is greater than a given threshold. A straightforward approach to answering this question would be to draw random samples from the input distribution, evaluate the model for each of these samples, and count how often the quantity of interest exceeds the threshold. This method may not be practical for several reasons: the model may be computationally expensive to evaluate, the dimension of the problem may be too large to construct an accurate response surface approximation, or the true probability may be very small (rare events).

Building a response surface approximation in the active subspace can help resolve these issues, but will introduce some error into the results. In general, a sample of a response surface approximation contains error due to interpolation of the response surface as well as any discretization errors that were present in the computational model. It has been shown that the adjoint-based approach for a posteriori error estimation can be modified to estimate these sources of error [6, 7, 5] and to devise adaptive methods to effectively reduce the dominant components [3, 16]. In this paper, we are primarily concerned with the additional error in the response surface approximation due to the dimension truncation. A combined analysis of all sources of error is beyond the scope of this paper and may be the subject of future work.

Given a response surface approximation in the active subspace and a particular event of interest, we can provide upper and lower bounds on the probability of this event if we can bound the dimension truncation error for each sample of the response surface. These bounds on the probability are produced by considering three cases for each sample of the response surface approximation. Samples whose value lies outside of the event and whose error bounds do not cross the threshold value are considered "definitely out". Similarly, samples whose value lies inside of the event and whose error bounds do not cross the threshold value are considered "definitely in". The remaining samples are classified as "unsure" since the distance between their value and the threshold is smaller than the error bound. The lower bound on the probability is computed using only the samples that are "definitely in" the event. The upper bound is computed using the samples that are "definitely in" and those that are "unsure". If the dimension truncation error bounds are robust, then it is easy to show that these lower and upper bounds on the probability are also robust for the given set of samples.

3.1. A Statistical Approach. One approach for estimating the dimension truncation error is to use the variance in the inactive subspace. Chebyshev's inequality states that for a random variable X,

$$\mathbb{P}\left(|X - \mathbb{E}(X)| \geq a\right) \leq \frac{\mathbb{V}\mathrm{ar}(X)}{a^2}. \tag{3.1}$$

Let $X = f\left(W_1^T \boldsymbol{y} + W_2^T \boldsymbol{z}\right)$ where the active variable is fixed. The expectation is then given by Equation 2.10 and the variance is easily estimated using the same sampling used to build $g(\boldsymbol{y})$. Hence, Equation 3.1 can be rewritten as

$$\mathbb{P}_{\boldsymbol{z}}\left(|f\left(W_1 \boldsymbol{y} + W_2 \boldsymbol{z}\right) - g(\boldsymbol{y})| \geq a\right) \leq \frac{\sigma_{\boldsymbol{z}}^2(\boldsymbol{y})}{a^2} \tag{3.2}$$

where $\sigma_{\boldsymbol{z}}^2(\boldsymbol{y})$ is the variance with respect to the inactive variable for a particular value of the active variable. The range of probabilities is constructed by first choosing a confidence coefficient $1 - \alpha$. For each sample, let a be the distance between $g(\boldsymbol{y})$ and the threshold. If

$\sigma_{\boldsymbol{z}}^2(\boldsymbol{y})/a^2 \leq 1-\alpha$, then the probability that the surrogate evaluation incorrectly placed the sample is low. Otherwise, the sample is treated like a samples whose error bound intersects the threshold.

This statistical approach provides an easily computed probabilistic error bound based on the variation in the inactive subspace. It is based on the assumption that the distribution of the variation is an aleatoric random variable which justifies the probabilistic interpretation. In this paper, we are also interested in an epistemic, or reducible, interpretation of the variation in the inactive subspace since the variation is bounded for any choice of active subspace and can be reduced by increasing the dimension of the active subspace, provided the associated eigenvalues of the additional dimensions are non-zero.

3.2. An Optimization-Based Approach. A second approach for estimating the error from the dimension reduction is to build a second response surface for dimension truncation error for each point in the active subspace. Given $\boldsymbol{y}$ in the active subspace, the goal is to find the point in the inactive subspace which results in the greatest deviation from the average

$$\boldsymbol{z}^*(\boldsymbol{y}) = \arg\max_{\boldsymbol{z}} \frac{1}{2}\left(f\left(W_1\boldsymbol{y} + W_2\boldsymbol{z}\right) - g(\boldsymbol{y})\right)^2. \tag{3.3}$$

The error bound for a particular $\boldsymbol{y}$ is then set to be $|f\left(W_1\boldsymbol{y} + W_2\boldsymbol{z}^*\right) - g(\boldsymbol{y})|$. Any optimization technique can be used to find $\boldsymbol{z}$. For the results in Section 4, the trust region conjugate gradient method described in Algorithm 4.1 of [22] was applied to the inactive variable. It should be noted that this algorithm effectively finds local maxima along the inactive subspace for given values in the active subspace. In order to make this search more global, multiple initial guesses for each active variable can be used. In fact, the random sampling used in Algorithm 2 can serve as initial guesses for the optimization algorithm. Thus, the optimization approach requires more model evaluations than the statistical approach, but it should also provide more robust and reliable bounds on the variation in the inactive subspace.

4. Results.

4.1. Laminar 1-D Channel Flow Problem. The first model examined is a steady-state, two-dimensional channel flow problem. In this problem, an incompressible fluid is flowing between two parallel plates down an infinitely long channel. No slip conditions are applied along the walls of the channel and a pressure drop is imposed along the channel to drive the flow. The momentum and continuity equations for this problem are

$$\rho\boldsymbol{u}\cdot\nabla\boldsymbol{u} - \mu\nabla^2\boldsymbol{u} + \nabla\left(p + p_0\right) = 0 \tag{4.1}$$

$$\nabla\cdot\boldsymbol{u} = 0 \tag{4.2}$$

where the input model parameters are the fluid viscosity μ, density ρ, and the pressure drop $\frac{\partial p_0}{\partial x}$. For convenience, these inputs are written in a single input vector

$$\boldsymbol{x} = \begin{bmatrix}\mu & \rho & \frac{\partial p_0}{\partial x}\end{bmatrix}^T. \tag{4.3}$$

As shown in Figure 4.1, the fully formed velocity profile is parabolic and can be found analytically to be

$$u(y) = \frac{1}{2}H^2\frac{1}{\mu}\frac{\partial p_0}{\partial x}\left(1 - \left(\frac{y}{H}\right)^2\right) \tag{4.4}$$

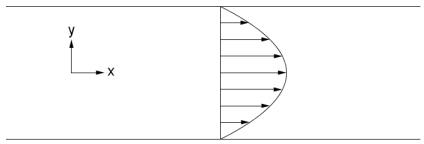
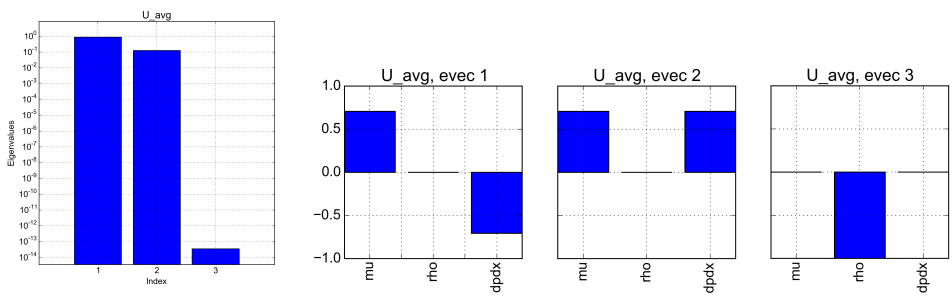


Fig. 4.1: The set up of the one-dimensional channel flow problem. The fully-formed velocity profile is parabolic and parallel to the channel. Integrating across the channel yields the average flow velocity.

Integrating this velocity profile across the channel (assume the channel has width $2H$) yields the the average flow velocity

$$u_{avg}(\boldsymbol{x}) = \frac{2}{3}H^3\frac{1}{\mu}\frac{\partial p_0}{\partial x}. \tag{4.5}$$

This will be the quantity of interest that is examined for this problem.



(a) Eigenvalues (b) Eigenvectors

Fig. 4.2: The eigenvalues and eigenvectors resulting from the applying Algorithm 1 to the channel flow problem.

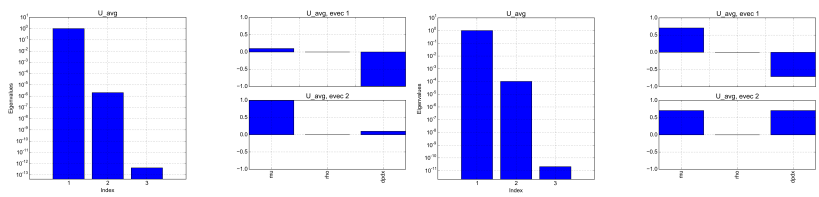
In order to restrict the problem to laminar flow, each of the parameters were chosen from the interval $[0.5, 10]$ with uniform probability. The eigenvalues and eigenvectors calculated in Algorithm 1 and which are used to define the active and inactive subspaces are shown in Figure 4.2.

First, we see that the third eigenvalue is nearly zero which implies that its associated eigenvector has very little impact on the quantity of interest. The entries of this eigenvector define a direction in the parameter space which is nearly equivalent to changes only in the density. Thus, variations in the physical parameter ρ have essentially no impact on the quantity of interest. This conclusion is obvious upon examining Equation 4.5. However, results such as this may not be obvious in problems without analytic solutions but can provide valuable insight into the behavior of the model.

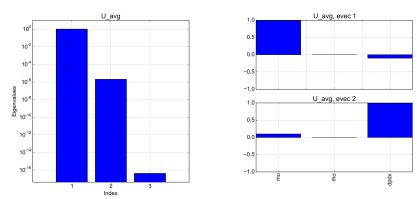
Next, we consider the two eigenvectors which do contribute to the average flow velocity. The eigenvector associated with the dominant eigenvalue defines a direction in the parameter space where changes in the viscosity are matched by equal but opposite changes in the pressure drop. Relating this to the analytic quantity of interest, we see an inverse relationship

between these two parameters. Thus, it is logical that the most dramatic change in u_{avg} is in the direction of the first eigenvalue. The second eigenvector represents equal changes in the viscosity and the pressure drop. Traveling along this direction in the parameter space can impact the quantity of interest (consider adding 1 to each term when $\mu = 1$ and $\frac{\partial p_0}{\partial x} = 2$). However, this change will not be as dramatic as changes resulting from moving along the first eigenvector. Furthermore, there are points in the parameter space where movement along the second eigenvector result in no change in the average flow velocity ($\mu = \frac{\partial p_0}{\partial x}$).

By focusing on smaller regions of the parameter domain, further insight into the behavior of the active subspaces is obtained. Recall that the original problem set up selected input values between 0.5 and 10.0 from a uniform distribution. Figure 4.3 contain the eigenvalues and two dominant eigenvectors for three small parameter ranges. The first subset uses viscosity values in a small interval around 10 and pressure drop values in a small interval around 1. The second subset uses a small interval around 1 for both μ and $\frac{\partial p_0}{\partial x}$. The last subset uses viscosity values around 1 and pressure drop values around 10. Since the quantity of interest is independent of the fluid density, ρ is randomly selected from the full domain in all three plots.



(a) $\mu \approx 10$ and $\frac{\partial p_0}{\partial x} \approx 1$. (b) $\mu \approx 1$ and $\frac{\partial p_0}{\partial x} \approx 1$.



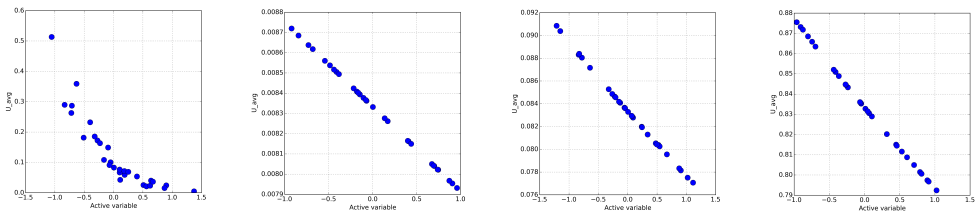
(c) $\mu \approx 1$ and $\frac{\partial p_0}{\partial x} \approx 10$.

Fig. 4.3: The eigenvalues and two dominant eigenvectors for three small regions of the domain. Note that ρ can be selected from its full domain since it does not influence the quantity of interest.

The first thing to notice is that for all three domains, the gap between the first and second eigenvalue is much larger than in the case of the full parameter domain. This implies that for all three cases, the first eigenvector will account for a much larger portion the total variation in the quantity of interest. However, each of the domains have different eigenvectors. For large μ and small $\frac{\partial p_0}{\partial x}$, the dominant eigenvector is weighted heavily in the direction of the pressure drop. Given the inverse relationship of these two parameters in Equation 4.5, we would expect the output to be more sensitive to variations in the smaller parameter. The situation is reversed in the case of a small viscosity and large pressure drop. Finally, when the two parameters are approximately equal, the result mimics the full domain.

Figure 4.4 contains scatter plots of the quantity of interest against the first active vari-

able for the full parameter domain and the three small subsets of the domain discussed above. The full domain produces a relatively strong relationship between the average flow velocity and the active variable. However, there is some variation which is due to influence from the other eigenvectors. The three small domains result in extremely tight linear relationships between the output and the active variables. Constructing the active subspaces attempts to find a linear relationship between the inputs which can accurately describe the global behavior of the model output. This linear approximation is strong over small regions, but will start to break down as the domain is expanded. Furthermore, additional linear relationships (that is, the other eigenvectors) serve as correction terms. In the case of small domains, very little correction is necessary as shown by the rapid decay in the eigenvalues. However, over the full domain, the correction is significant as is seen by the variation of the output with respect to just the first active variable.



$\mu \approx 10$ and $\frac{\partial p_0}{\partial x} \approx 1$ $\mu \approx 1$ and $\frac{\partial p_0}{\partial x} \approx 1$ $\mu \approx 1$ and $\frac{\partial p_0}{\partial x} \approx 10$

Fig. 4.4: Scatter plots of the average flow velocity for the full parameter range and three subsets of the parameter domain.

4.2. Magnetohydrodynamics Hartmann Flow Problem. The Hartmann flow problem is similar to the channel flow problem discussed above with the addition of a transverse magnetic field acting on the ionized fluid. The magnetic field acts as a force resisting the fluid flow so that the momentum equation includes a Lorentz force term. This term exerts a force whenever the magnetic field lines are bent (non-zero curl).

$$\rho \boldsymbol{u} \cdot \nabla \boldsymbol{u} - \mu \nabla^2 \boldsymbol{u} + \nabla (p + p_0) = -\frac{1}{\mu} \boldsymbol{B} \times (\nabla \times \boldsymbol{B}) \tag{4.6}$$

$$\nabla \cdot \boldsymbol{u} = 0 \tag{4.7}$$

The fluid flow induces a magnetic field in the horizontal direction. The induction equation for the magnetic field is given by

$$\boldsymbol{u} \cdot \nabla \boldsymbol{B} = \frac{\eta}{\mu} \nabla^2 \boldsymbol{B} + \boldsymbol{B} \cdot \nabla \boldsymbol{u} \tag{4.8}$$

$$\nabla \cdot \boldsymbol{B} = 0. \tag{4.9}$$

The model inputs are the fluid viscosity μ, density ρ, pressure drop $\frac{\partial p_0}{\partial x}$, magnetic resistivity η, and applied magnetic field B_0. As before, the input parameters are written into a single input vector

$$\boldsymbol{x} = \begin{bmatrix} \mu & \rho & \frac{\partial p_0}{\partial x} & \eta & B_0 \end{bmatrix}^T. \tag{4.10}$$

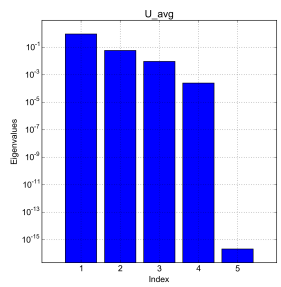
For this study, the parameters were allowed to range between 0.5 and 3.0 with a uniform distribution applied.

Similar to the channel flow problem, the Hartmann problem has an analytic solution. However, solutions were obtained using Drekar [23]. There are several quantities of interest that can be extracted from this problem. The two that are examined here are the average flow velocity

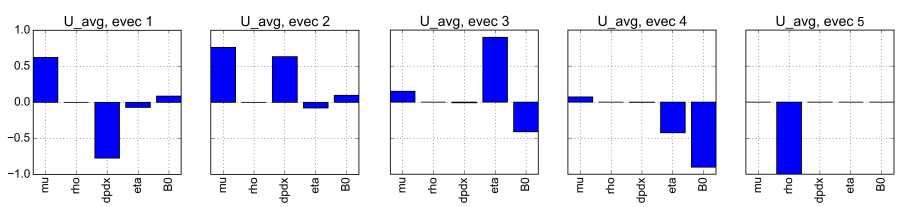
$$u_{avg}(\boldsymbol{x}) = \frac{\partial p_0}{\partial x} \frac{B_0\sqrt{\eta\mu}\coth\left(\frac{B_0}{\sqrt{\eta\mu}}\right) - \eta\mu}{2B_0\mu^2} \tag{4.11}$$

and the induced magnetic field

$$B_{ind}(\boldsymbol{x}) = \frac{\partial p_0}{\partial x} \frac{\mu_0\left(B_0 - 2\sqrt{\eta\mu}\tanh\left(\frac{B_0}{2\sqrt{\eta\mu}}\right)\right)}{2B_0^2}. \tag{4.12}$$



(a) Eigenvalues

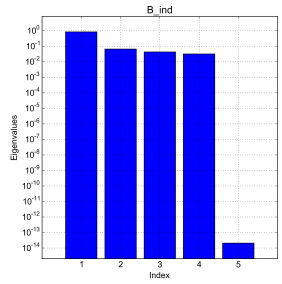


(b) Eigenvectors

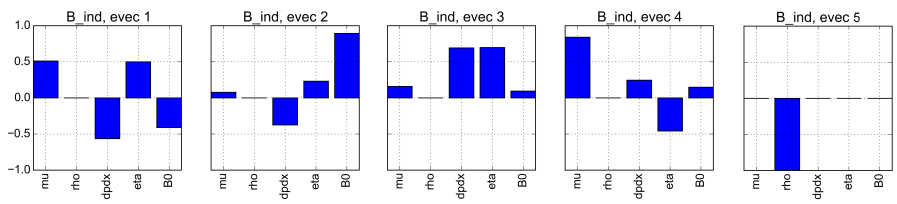
Fig. 4.5: The eigenvalues and eigenvectors for the average flow velocity quantity of interest.

Figure 4.5 provides the eigenvalues and eigenvectors associated the average flow velocity. As with the channel flow problem, the fifth eigenvalue is nearly zero and its associated eigenvector corresponds directly to the fluid density. This lack of influence is again apparent from examination of Equation 4.11. Additionally, the first two eigenvectors closely resemble the first two eigenvectors in the channel flow problem. The entries associated with the magnetic inputs are small. The next two eigenvectors are driven almost exclusively by the magnetic inputs.

Next, we examine the behavior of the induced magnetic field. The eigenvalue and eigenvectors are given by Figure 4.6. In this case, we do not see much decay in the eigenvalues suggesting the lack of a useful active subspace. The first eigenvector contains contributions from all parameters (except for ρ, which we no has no contribution to the quantity of



(a) Eigenvalues



(b) Eigenvectors

Fig. 4.6: The eigenvalues and eigenvectors for the induced magnetic field quantity of interest.

interest) in approximately equal magnitudes. The eigenvalues associated with the next three eigenvectors are nearly identical implying that each contributes comparable variation to the output.

4.3. SARANS Approximation of Fully-Developed 3-D Flow in Pipe. In this section, we consider fully developed flow in a three-dimensional circular pipe. The flow is considered fully developed when the radial velocity profile does not change with axial location in the pipe. This condition is achieved through periodic inflow-outflow boundary conditions and applying a momentum source term to force the flow in the axial direction.

Drekar::CFD solutions were obtained by solving an SARANS formulation with stabilized finite elements using a direct-to-steady-state approach on a mesh containing 168,000 elements with 1M unknowns [24]. The tube diameter is ($D = 0.01\ m$) and pipe axis is aligned with the x-direction. Two views of a slightly coarser computational mesh are shown in Figure 4.7.

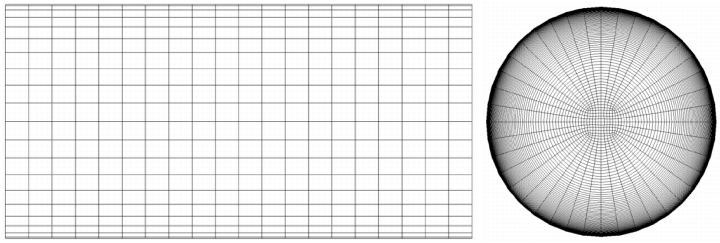


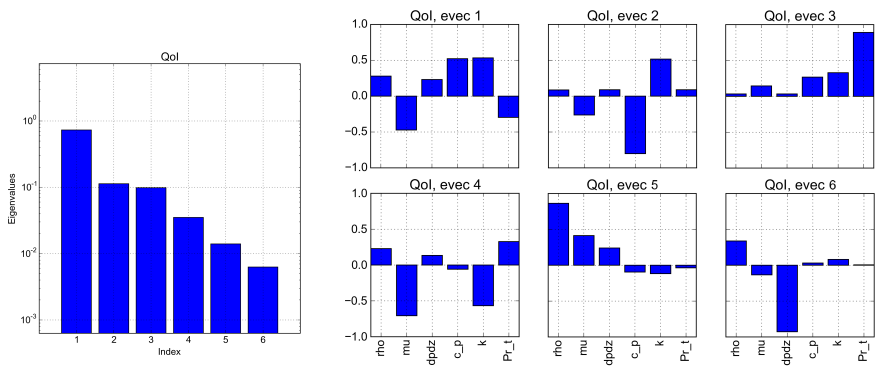
Fig. 4.7: Two cut-away views of a 62,000 element pipe mesh.

We assume that six of the model parameters are uniformly distributed over parameters ranges given by:

$$\rho \in [0.5, 2], \quad \mu \in [5\text{E-}7, 5\text{E-}6], \quad -\frac{\partial p}{\partial z} \in [2.5, 7.5],$$

$$C_p \in [0.1, 10], \quad \kappa \in [5\text{E-}7, 5\text{E-}6], \quad Pr_t \in [0.1, 10],$$

where C_p is the specific heat, κ is the thermal conductivity, and Pr_t is the turbulent Prandtl number. This study considers the wall temperature at the exit of the pipe, T_w, as the QoI. We use a Latin hypercube strategy to produce 80 samples over the range of the input parameters and use Drekar::CFD to compute the QoI and the gradient of the QoI at these sample points. Each of these function evaluations is a direct-to-steady-state solve of the 3D coupled SARANS turbulent fluid flow and heat transfer problem. In Figure 4.8, we see a



(a) Eigenvalues (b) Eigenvectors

Fig. 4.8: The eigenvalues and eigenvectors for the computed quantity of interest.

relatively smooth decay of the eigenvalues, but no obvious choice of truncation for an active subspace.

4.4. Bounds on Probabilities and Distributions for Hartmann Problem. In this section, we consider the Hartmann problem described in Section 4.2 and apply the statistical and optimization-based approaches for estimating the dimension truncation error to provide upper and lower bounds on probabilistic quantities of interest. The response surface approximations for the model and the bounds are constructed using radial basis functions on the active variables. Recall that these active variables are constructed using the eigenvectors of the active subspace analysis. Thus, the eigenvalues can be used in determining the length scales of the radial basis functions along each axes in the reduced space. For each quantity of interest, we choose threshold values so that the probability of the response exceeding these values is small. Since both quantities of interest have analytic solutions, the true probabilities can be computed using a large number of Monte Carlo samples over the full parameter space.

In Table 4.1, we give the estimates of the probability of each quantity of interest exceeding a particular threshold as well as the bounds on the probabilities using both error bound approaches for reduced subspaces of varying dimension ($n = 1, 2, 3$). The objective

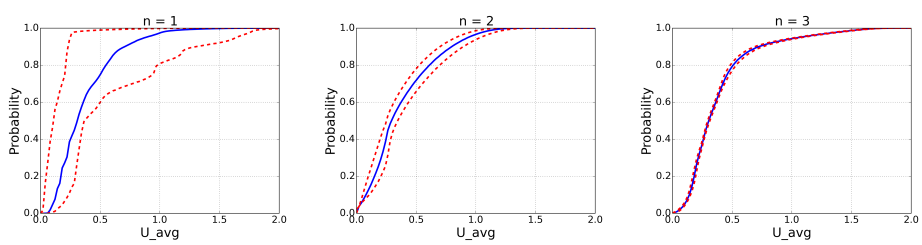
$P[u_{avg} > 1.0] = 0.034$			
	$n = 1$	$n = 2$	$n = 3$
Response Surface Approximation	0.027	0.037	0.033
Optimization Bound	$[0.003, 0.142]$	$[0.026, 0.053]$	$[0.032, 0.034]$
Variance Bound ($\alpha = 0.95$)	$[0.001, 0.168]$	$[0.017, 0.064]$	$[0.031, 0.035]$

$P[B_{ind} > 0.2] = 0.011$			
	$n = 1$	$n = 2$	$n = 3$
Response Surface Approximation	0.008	0.008	0.012
Optimization Bound	$[0.001, 0.068]$	$[0.002, 0.032]$	$[0.010, 0.017]$
Variance Bound ($\alpha = 0.95$)	$[0.001, 0.094]$	$[0.001, 0.048]$	$[0.006, 0.029]$

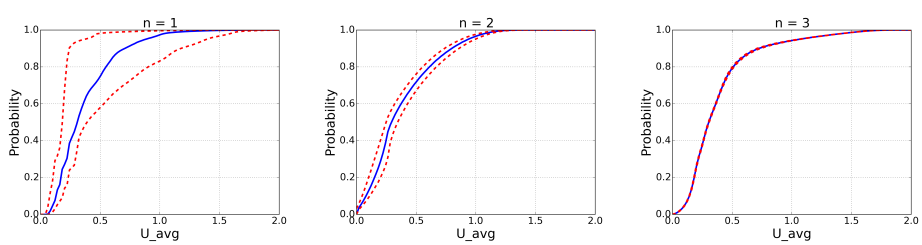
Table 4.1: Probability bounds using the optimization and variance bounds for increasing active subspace dimension.

here is not to select the optimal dimension truncation, but to account for the error induced by the dimension truncation.

Given the error bounds using either statistical or the optimization-based approach, it is relatively easy to compute bounds for a range of probability levels for each quantity of interest. Therefore, we can easily generate cumulative distribution functions (CDF) with upper and lower bounds. In Figures 4.9 and 4.10, we plot the results for the average x-velocity and the average induced magnetic field respectively.

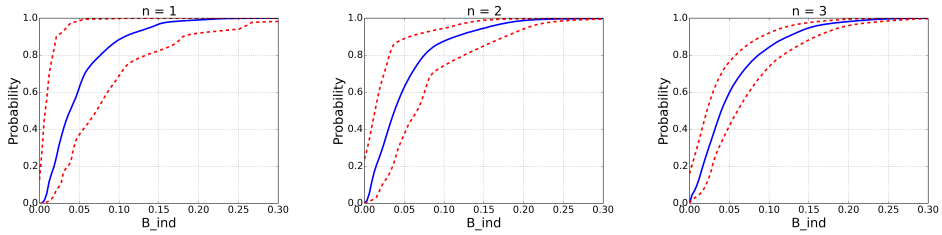


(a) Statistical approach

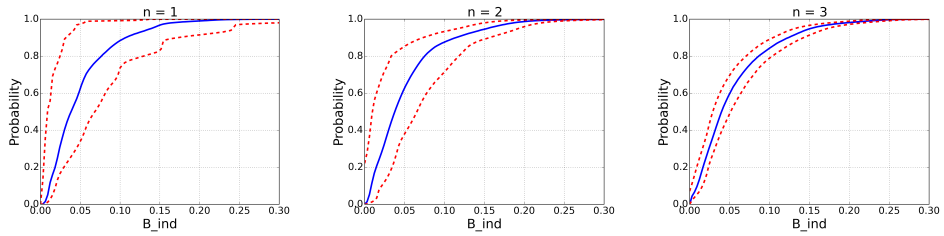


(b) Optimization approach

Fig. 4.9: Bounds on CDF of average x-velocity using the statistical approach (top) and the optimization-based approach (bottom) for $n = 1, 2, 3$.



(a) Statistical approach



(b) Optimization approach

Fig. 4.10: Bounds on CDF of average induced magnetic field using the statistical approach (top) and the optimization-based approach (bottom) for $n = 1, 2, 3$.

In both Table 4.1 and Figures 4.9 and 4.10, we see that introducing another dimension to the active subspace tightens the bounds on the error. Recall that the error in the n-dimensional response surface due to the dimension truncation is on the order of the sum of the last $m - n$ eigenvalues. Comparing the two quantities of interest, we see the eigenvalues from the average flow velocity decay much more rapidly than those from the induced magnetic field. Thus, for the same value of n, the error bounds are much tighter for u_{avg}.

5. Conclusion. We have provided a concise overview of active subspaces and demonstrated the process of identifying an active subspace. We applied this methodology to a set of problems in CFD and MHD and observed a small reduction in dimension for each problem. We also showed how to use optimization and statistical approaches to estimate the dimension truncation error and showed how these estimates can be used to compute upper and lower bounds on probabilistic quantities of interest. Future work will involve problems with a larger number of parameters and an approach to estimate the total error (discretization, interpolation, dimension reduction, etc.) in a sample of a response surface in the active subspace.

REFERENCES

[1] I. Babuška, F. Nobile, and R. Tempone, *A stochastic collocation method for elliptic partial differential equations with random input data*, SIAM J. Numer. Anal., 45 (2007), pp. 1005–1034 (electronic).

[2] J. Breidt, T. Butler, and D. Estep, *A computational measure theoretic approach to inverse sensitivity problems I: Basic method and analysis*, SIAM J. Numer. Analysis, 49 (2012), pp. 1836–1859.

[3] C. Bryant, S. Prudhomme, and T. Wildey, *Error decomposition and adaptivity for response surface approximations from PDEs with parametric uncertainty.* To appear in SIAM/ASA J. Uncert. Quant., 2015.

[4] H.-J. Bungartz and M. Griebel, *Sparse grids*, Acta Numerica, 13 (2004), pp. 147–269.

[5] T. Butler, P. Constantine, and T. Wildey, *A posteriori error analysis of parameterized linear systems using spectral methods*, SIAM. J. Matrix Anal. Appl., 33 (2012), pp. 195–209.

[6] T. Butler, C. Dawson, and T. Wildey, *A posteriori error analysis of stochastic spectral methods*, SIAM J. Sci. Comput., 33 (2011), pp. 1267–1291.

[7] ———, *Propagation of uncertainties using improved surrogate models*, SIAM/ASA Journal on Uncertainty Quantification, 1 (2013), pp. 164–191.

[8] D. Cacuci, *Sensitivity and Uncertainty Analysis: Theory*, vol. I, Chapman & Hall/CRC, 1997.

[9] P. G. Constantine, *Active Subspaces: Emerging Ideas for Dimension Reduction in Parameter Studies*, SIAM, 2014.

[10] P. G. Constantine, M. Emory, J. Larsson, and G. Iaccarino, *Exploiting Active Subspaces to Quantify Uncertainty in the Numerical Simulation of the HyShot II Scramjet*, arXiv, (2014).

[11] P. G. Constantine, B. Zaharatos, and M. Campanelli, *Discovering an active subspace in a single-diode solar cell model*, arXiv, (2014).

[12] D. Estep and D. Neckels, *Fast and reliable methods for determining the evolution of uncertain parameters in differential equations*, J. Comput. Physics, 213 (2006), pp. 530–556.

[13] B. Ganis, H. Klie, M. F. Wheeler, T. Wildey, I. Yotov, and D. Zhang, *Stochastic collocation and mixed finite elements for flow in porous media*, Comput. Methods Appl. Mech. Engrg, 197 (2008), pp. 3547 – 3559. Stochastic Modeling of Multiscale and Multiphysics Problems.

[14] T. Gerstner and M. Griebel, *Dimension-adaptive tensor-product quadrature*, Computing, 71 (2003), pp. 65–87.

[15] R. Ghanem and P. Spanos, *Stochastic Finite Elements: A Spectral Approach*, Springer Verlag, New York, 2002.

[16] J. Jakeman and T. Wildey, *Enhancing adaptive sparse grid approximations and improving refinement strategies using adjoint-based a posteriori error estimates*, Journal of Computational Physics, 280 (2015), pp. 54 – 71.

[17] C. Lanczos, *Linear Differential Operators*, Dover Publications, 1997.

[18] T. W. Lukaczyk, P. G. Constantine, F. Palacios, and J. J. Alonso, *Active subspaces for shape optimization*, in 10th AIAA Multidisciplinary Design Optimization Conference, American Institute of Aeronautics and Astronautics, 2014.

[19] X. Ma and N. Zabaras, *An adaptive hierarchical sparse grid collocation algorithm for the solution of stochastic differential equations*, J. Comput. Phys., 228 (2009), pp. 3084–3113.

[20] G. I. Marchuk, *Adjoint equations and analysis of complex systems*, Kluwer, 1995.

[21] G. I. Marchuk, V. I. Agoshkov, and V. P. Shutyaev, *Adjoint Equations and Perturbation Algorithms in Nonlinear Problems*, CRC Press, Boca Raton, FL, 1996.

[22] J. Nocedal and S. J. Wright, *Numerical Optimization*, Springer, 2006.

[23] R. Pawlowski, J. Shadid, T. Smith, E. C. Cyr, and P. Weber, *Drekar::CFD-a turbulent fluid-flow and conjugate heat transfer code: Theory manual version 1.0*, Tech. Rep. SAND2012-2697, Sandia National Laboratories, March 2012.

[24] J. N. Shadid, T. M. Smith, E. C. Cyr, T. M. Wildey, and R. P. Pawlowski, *Stabilized FE Simulation of Prototype Thermal-Hydraulics Problems with Integrated Adjoint-based Capabilities*. Journal of Computational Physics, 2015.

[25] X. Wan and G. Karniadakis, *Beyond Wiener-Askey Expansions: Handling Arbitrary PDFs*, Journal of Scientific Computing, 27 (2006), pp. 455–464.

[26] D. Xiu and G. Karniadakis, *The Wiener-Askey polynomial chaos for stochastic differential equations*, SIAM J. Sci. Comput., 24 (2002), pp. 619–644.

A TIME-PARALLEL METHOD FOR THE SOLUTION OF PDE-CONSTRAINED OPTIMIZATION PROBLEMS

MONA HAJGHASSEM*, ERIC C. CYR†, AND DENIS RIDZAL‡

Abstract. We study a time-parallel approach to solving quadratic optimization problems with linear time-dependent partial differential equation (PDE) constraints. These problems arise in formulations of optimal control, optimal design and inverse problems that are governed by parabolic PDE models. They may also arise as subproblems in algorithms for the solution of optimization problems with nonlinear time-dependent PDE constraints, e.g., in sequential quadratic programming methods. We apply a piecewise linear finite element discretization in space to the PDE constraint, followed by the Crank-Nicolson discretization in time. The objective function is discretized using finite elements in space and the trapezoidal rule in time. At this point in the discretization, auxiliary state variables are introduced at each discrete time interval, with the goal to enable: (i) a decoupling in time; and (ii) a fixed-point iteration to recover the solution of the discrete optimality system. The fixed-point iterative schemes can be used either as preconditioners for Krylov subspace methods or as smoothers for multigrid (in time) schemes. We present promising numerical results for both use cases.

1. Introduction. Optimal control problems governed by time-dependent partial differential equations (PDEs) lead to large-scale optimization problems. The cost of numerically solving these problems is proportional to the size of the discretized time domain. In most numerical approaches the time discretization serializes the solution process, and introduces a bottleneck that cannot be overcome by additional parallelization in space. To speed up the solution process for linear-quadratic optimal control problems governed by parabolic PDEs, we study a time-domain decomposition approach with the potential to introduce time parallelism into the optimization algorithm. Parallelization in time approximates parts of the solution later in time simultaneously with the parts of the solution earlier in time.

In Gander's review article on parallel time integration [4], time-parallel methods are classified into four groups: methods based on multiple shooting; methods based on domain decomposition and waveform relaxation; space-time multigrid methods; and direct time-parallel methods. Our approach is based on domain decomposition. However, we note from the onset that our goal is the iterative solution of optimality systems that comprise discrete PDE systems, and not the solution of discrete PDE systems on their own as in [4] and the references cited therein. This introduces opportunities for time parallelization that have not been explored in the literature. Our work is closely related to the work of Heinkenschloss [5] and Comas [3], who consider time-domain decomposition (and multiple shooting) methods for optimal control problems to reduce storage demands imposed by the discretization of the state variables. In contrast, our work focuses on preconditioners that are appropriate for time parallelization. In principle, it is possible to balance storage reduction and time parallelization in a single scheme, which we plan to study in the future.

To streamline the presentation, we consider the Neumann boundary control of the one-dimensional linear heat equation. For the spatial discretization of the heat equation we use piecewise linear finite elements. For the time discretization we use the Crank-Nicolson method. The objective function is discretized using finite elements in space and the trapezoidal rule in time. At this point of the discretization process, we introduce auxiliary state variables $\bar{u}_i$ for each time interval, corresponding to the original discrete state variables u_i, and impose the time-continuity constraint equations $\bar{u}_i = u_i$. Additionally, the auxiliary variables $\bar{u}_i$ are incorporated into the objective function. The first-order necessary and suffi-

*University of Maryland Baltimore County, mona4@umbc.edu
†Sandia National Laboratories, Computational Mathematics, eccyr@sandia.gov
‡Sandia National Laboratories, Optimization and Uncertainty Quantification, dridzal@sandia.gov

cient optimality conditions for the discretized problem form a linear system $\mathbf{Ax} = \mathbf{b}$, where $\mathbf{x}$ is a vector composed of the state variables, auxiliary state variables, control variables, the Lagrange multipliers associated with the state equation and the Lagrange multipliers associated with the continuity equations. With a suitable ordering of the necessary and sufficient optimality conditions, we arrive at a block tridiagonal system of optimality conditions, presented in Section 2. The block tridiagonal structure motivates fixed-point iterations based on matrix splittings, which are discussed in Section 3. For instance, the optimality system can be solved using the GMRES method and time-parallel fixed-point preconditioners, such as the Jacobi scheme or the red-black Gauss-Seidel scheme. Fixed-point iterations can also be used as smoothers in the development of new multigrid in time techniques. In Section 4 we present promising numerical results for both use cases. We demonstrate the potential effectiveness of the Jacobi scheme as a time-parallel preconditioner for linear systems arising in, e.g., inexact sequential quadratic programming (SQP) methods. We also demonstrate the smoothing properties of Jacobi and Gauss-Seidel iterations, with a future goal of incorporating them in a multigrid in time scheme.

2. Neumann control of the one-dimensional heat equation. We consider the optimization problem

$$\min \frac{1}{2}\int_0^T z^2(t)dt + \frac{\alpha}{2}\int_0^T \int_0^1 (u(t,x) - s(t,x))^2 dxdt \tag{2.1}$$

governed by the one-dimensional linear heat equation

$$\frac{\partial}{\partial t}u(t,x) - \frac{\partial^2}{\partial x^2}u(t,x) = f(t,x) \qquad t \in (0,T),\ x \in (0,1), \tag{2.2a}$$

$$\frac{\partial}{\partial x}u(t,0) = z(t), \qquad t \in (0,T), \tag{2.2b}$$

$$\frac{\partial}{\partial x}u(t,1) = r(t), \qquad t \in (0,T), \tag{2.2c}$$

$$u(0,x) = u_0(x), \qquad x \in (0,1). \tag{2.2d}$$

The functions u and z are called state and control, respectively. The above problem is posed in Hilbert space, as described in [5]. We assume that (2.2) admits a unique solution u for every control z and that the above optimal control problem has a solution, see [3, 5].

We discretize the state equation in space by applying piecewise linear finite elements with N_x spatial intervals on the $[0,1]$ interval. We arrive at the semi-discrete problem

$$\min \frac{1}{2}\int_0^T z^2(t)dt + \frac{\alpha}{2}\int_0^T \|u(t) - s(t)\|^2 dt \tag{2.3}$$

subject to

$$M\frac{\partial}{\partial t}u(t) + Su(t) = f(t) + Wz(t) \qquad t \in (0,T), \tag{2.4a}$$

$$u(0) = u_0, \tag{2.4b}$$

where M and S are the mass and stiffness matrices, respectively, of size $(N_x+1)\times(N_x+1)$, $f(t)$ is an (N_x+1)-vector that includes the Neumann boundary term $r(t)$, and W is an $(N_x+1)\times 1$ operator that applies the control as the Neumann condition at $x=0$. We let $\|\cdot\|$ denote a discrete norm over space, defined in this case as $\|u\|^2 = u^T M u$. To avoid vector notation we introduce a hierarchical notation for functions, based on the space or

space-time discretization. We write $w(x,t)$ for a continuous function, $w(t)$ for its space discretization, and w_i for its space-time discretization, where i is a time index.

We apply the θ-method to discretize in time the semi-discrete constraint equation (2.4); in numerical experiments, we will use $\theta = 1/2$, i.e., the Crank-Nicolson method. We apply the trapezoidal rule to approximate the time integrals in the objective function (2.3). The time domain is divided into N intervals $[t_i, t_{i+1}]$, $i = 0, 1, ..., N-1$. At the time domain interfaces, auxiliary variables $\bar{u}_i$, $i = 1, ..., N-1$, are introduced to enforce time continuity of the state in the time-domain decomposition process. Additionally, these variables are incorporated in the objective function, and used simultaneously with the state variables u_i. The control variables $\bar{z}_i$ are scalar quantities, defined in each subinterval $[t_i, t_{i+1}]$. The fully discretized optimal control problem can be written as

$$\begin{aligned} \min J(u, \bar{u}, \bar{z}) = & \frac{\Delta t}{2} \sum_{i=0}^{N-1} (\bar{z}_i)^2 \\ & + \frac{\alpha \Delta t}{4} \sum_{i=0}^{N-1} (\|\bar{u}_i - s_i\|^2 + \|u_{i+1} - s_{i+1}\|^2) \end{aligned} \tag{2.5}$$

subject to

$$\begin{aligned} M \frac{u_{i+1} - \bar{u}_i}{\Delta t} = & - S(\theta u_{i+1} + (1-\theta)\bar{u}_i) + \theta(f_{i+1} + W\bar{z}_i) + (1-\theta)(f_i + W\bar{z}_i) \\ \bar{u}_i = u_i \qquad & i = 0, 1, \ldots, N-1, \end{aligned} \tag{2.6}$$

where Δt is the time step. By rearranging the constraint equation we obtain:

$$\begin{aligned} (M + \Delta t \theta S) u_{i+1} = & (M - \Delta t(1-\theta)S)\bar{u}_i + \Delta t(W\bar{z}_i + \theta f_{i+1} + (1-\theta) f_i) \\ \bar{u}_i = u_i \qquad & i = 0, 1, \ldots, N-1. \end{aligned} \tag{2.7}$$

Note that the minimizer of the time-domain decomposition formulation is also a minimizer of the discretized continuous time formulation. To see this let u_i^*, $\bar{u}_i^*$ and $\bar{z}_i^*$ satisfy Eq. (2.5) subject to Eq. (2.7). Then eliminating the auxiliary variable $\bar{u}_i^*$ using the continuity condition, the state constraint implies

$$(M + \Delta t \theta S) u_{i+1} = (M - \Delta t(1-\theta)S) u_i + \Delta t \bar{z}_i + \Delta t(\theta f_{i+1} + (1-\theta) f_i). \tag{2.8}$$

This is an approximation of Eq. (2.4) using the θ-method. Similarly, using the continuity condition to eliminate $\bar{u}_i^*$ from Eq. (2.5) gives the trapezoidal-rule approximation of the objective function from Eq. (2.3).

The motivation for introducing the continuity condition is twofold. First, including it in the discretization is more general than enforcing it by construction. Given an implementation of the discontinuous case it is easy to construct the continuous case. Second, explicit treatment of the continuity condition as a state equation exposes the condition to the optimization algorithm. This enables a lifted Newton method with respect to the time-continuous problem [1], which may (in future studies) provide additional algorithmic advantages.

The Lagrangian function for the above discretized problem is given by

$$\begin{aligned} L(u,\bar{u},\bar{z},\lambda,\bar{\lambda}) = & \frac{\Delta t}{2}\sum_{i=0}^{N-1}(\bar{z}_i)^2 \\ & + \frac{\alpha\Delta t}{4}\sum_{i=0}^{N-1}\left(\|\bar{u}_i - s_i\|^2 + \|u_{i+1} - s_{i+1}\|^2\right) \\ & + \sum_{i=0}^{N-1}\left(\lambda_{i+1}, K\bar{u}_i + \Delta t W\bar{z}_i + \Delta t(\theta f_{i+1} + (1-\theta)f_i) - Cu_{i+1}\right) \\ & + \sum_{i=0}^{N-1}\left(\bar{\lambda}_{i+1}, u_i - \bar{u}_i\right), \end{aligned} \tag{2.9}$$

where $C = M + \Delta t\theta S$ and $K = M - \Delta t(1-\theta)S$. The optimality conditions can be obtained by setting the partial derivatives of L with respect to $\bar{u}_i$, $\bar{z}_i$, λ_i, $\bar{\lambda}_i$, and u_i to zero. The partial derivatives of the Lagrangian read:

$$\begin{aligned} \frac{\partial L}{\partial \bar{u}_i} &= \frac{\alpha\Delta t}{2}M(\bar{u}_i - s_i) + K^T\lambda_{i+1} - \bar{\lambda}_{i+1}, && \text{for } i = 0\ldots N-1; \\ \frac{\partial L}{\partial \bar{z}_i} &= \Delta t(\bar{z}_i + W^T\lambda_{i+1}), && \text{for } i = 0\ldots N-1; \\ \frac{\partial L}{\partial \lambda_i} &= K\bar{u}_{i-1} + \Delta t W\bar{z}_{i-1} + \Delta t\theta f_i + (1-\theta)f_{i-1} - Cu_i, && \text{for } i = 1\ldots N; \\ \frac{\partial L}{\partial \bar{\lambda}_i} &= u_{i-1} - \bar{u}_{i-1}, && \text{for } i = 2\ldots N; \\ \frac{\partial L}{\partial u_i} &= \frac{\alpha\Delta t}{2}M(u_i - s_i) - C^T\lambda_i + \bar{\lambda}_{i+1}, && \text{for } i = 1\ldots N. \end{aligned} \tag{2.10}$$

Notice for brevity we have not been precise with respect to the fringe cases (e.g., $i = 0$, $i = N$).

The Lagrangian defined in Eq. (2.9) uses an identity matrix in space to enforce continuity in time (e.g., enforcing the constraint $\bar{u}_i = u_i$). This implies the adjoint equations correspond to

$$C^T\lambda_i - \bar{\lambda}_{i+1} = \frac{\alpha\Delta t}{2}M(u_i - s_i) \tag{2.11}$$

$$K^T\lambda_{i+1} - \bar{\lambda}_{i+1} = \frac{\alpha\Delta t}{2}M(\bar{u}_i - s_i). \tag{2.12}$$

The equations represent adjoint time continuity conditions. In the first equation the adjoint variables λ_i and $\bar{\lambda}_{i+1}$ have different scaling. This can be optionally rectified by changing the state continuity condition to $C(\bar{u}_i - u_i) = 0$. In the numerical results section this condition is used for the smoothing study, while the original condition $\bar{u}_i = u_i$ is used for the GMRES study.

The optimality conditions lead to a linear system, $\mathbf{Ax} = \mathbf{b}$. The equations and variables in the linear system can be ordered in a variety of ways. We take adjoint variables to be λ_i and $\bar{\lambda}_i$ instead of λ_{i+1} and $\bar{\lambda}_{i+1}$. We consider the following ordering for the vector $\mathbf{x}$,

$$\mathbf{x}^T = \begin{pmatrix} . & . & . & \bar{u}_i & \bar{z}_i & \lambda_{i+1} & \bar{\lambda}_{i+1} & u_{i+1} & . & . & . \end{pmatrix}. \tag{2.13}$$

This presents the ordering in the interior of the time domain focusing on the time interval $[t_i, t_{i+1}]$. Again for simplicity we ignore the ordering for $i = 0$ and $i = N$. The matrix $\mathbf{A}$ corresponding to the vector $\mathbf{x}$ is of size $N \times N$ blocks, and reads:

$$\mathbf{A} = \begin{pmatrix} D & U & & & \\ L & D & U & & \\ & \ddots & \ddots & \ddots & \\ & & L & D & U \\ & & & L & D \end{pmatrix}, \tag{2.14}$$

where

$$D = \begin{pmatrix} \frac{\alpha_1 \Delta t}{2} M & 0 & K^T & -I & 0 \\ 0 & \Delta t I & \Delta t W^T & 0 & 0 \\ K & \Delta t W & 0 & 0 & -C \\ -I & 0 & 0 & 0 & 0 \\ 0 & 0 & -C^T & 0 & \frac{\alpha_1 \Delta t}{2} M \end{pmatrix} \tag{2.15}$$

and

$$L = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & I \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}, \qquad U = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & I & 0 \end{pmatrix}. \tag{2.16}$$

The right-hand side vector $\mathbf{b}$ is:

$$\mathbf{b}^T = \begin{pmatrix} 0 & 0 & -F_1 & 0 & \frac{\alpha_1 \Delta t}{2} s_1 & . & . & . & 0 & 0 & -F_N & 0 & \frac{\alpha_1 \Delta t}{2} s_N \end{pmatrix}. \tag{2.17}$$

3. Time-domain decomposition preconditioners. As seen in Section 2, the system of optimality conditions for the time-dependent optimal control problem leads to a block tridiagonal linear system $\mathbf{Ax} = \mathbf{b}$. This system can be solved using the GMRES method of Saad and Schultz [8], combined with various preconditioners. In this section we discuss the construction of time-domain decomposition preconditioners based on matrix splitting methods, such as Jacobi or Gauss-Seidel.

We will use the 'right-preconditioned' form of $\mathbf{Ax} = \mathbf{b}$,

$$\mathbf{AP}^{-1}\mathbf{y} = \mathbf{b}, \qquad \mathbf{x} = \mathbf{P}^{-1}\mathbf{y}, \tag{3.1}$$

where $\mathbf{P}$ denotes the preconditioning matrix. We use the diag$(\cdot)$ operator to denote the $N \times N$ block diagonal matrix composed of its arguments. Similarly, we introduce operators subdiag$(\cdot)$ and supdiag$(\cdot)$, which denote $N \times N$ block subdiagonal and superdiagonal matrices composed of the operator arguments. We define the matrices:

$$\mathbf{D} = \text{diag}(D, \ldots, D); \quad \mathbf{L} = \text{subdiag}(L, \ldots, L); \quad \mathbf{U} = \text{supdiag}(U, \ldots, U);$$

and write the matrix $\mathbf{A}$ as their sum, $\mathbf{A} = \mathbf{L} + \mathbf{D} + \mathbf{U}$. Moreover, we can define preconditioners by zeroing some of the entries of the matrices $\mathbf{L}$ and $\mathbf{U}$, as follows. First, for a positive integer j, we introduce the matrices:

$$\mathbf{L}(j) = \text{subdiag}(L, \ldots, L, \underset{j}{0}, L, \ldots, L, \underset{2j}{0}, L, \ldots, L, \underset{3j}{0}, \ldots); \quad \text{and}$$
$$\mathbf{U}(j) = \text{supdiag}(U, \ldots, U, \underset{j}{0}, U, \ldots, U, \underset{2j}{0}, U, \ldots, U, \underset{3j}{0}, \ldots).$$

Second, recall that N denotes the total number of time steps, and let N_t denote the number of time subdomains. In $\mathbf{L}(j)$ and $\mathbf{U}(j)$ we zero the entries of $\mathbf{L}$ and $\mathbf{U}$ in locations that are the integer multiples of $j = \lfloor N/N_t \rfloor$. The locations j, $2j$, etc., indicate the boundaries of the time subdomains. In other words, they indicate the breaking of time continuity in the state variables. This leads to the definition of three preconditioners:

$$\begin{aligned}
\mathbf{P}_{\text{JAC}} &= \mathbf{L}(j) + \mathbf{D} + \mathbf{U}(j) : && \text{Jacobi;} \\
\mathbf{P}_{\text{FGS}} &= \mathbf{L} + \mathbf{D} + \mathbf{U}(j) : && \text{forward Gauss-Seidel; and} \\
\mathbf{P}_{\text{BGS}} &= \mathbf{L}(j) + \mathbf{D} + \mathbf{U} : && \text{backward Gauss-Seidel.}
\end{aligned}$$

For example, for the number of time steps $N = 10$ and the number of time sudomains $N_t = 2$, we have that $j = 5$, and the Jacobi preconditioner assumes the following form:

$$\mathbf{P}_{\text{JAC}} = \begin{pmatrix}
D & U & & & & & & & & \\
L & D & U & & & & & & & \\
 & L & D & U & & & & & & \\
 & & L & D & U & & & & & \\
 & & & L & D & & & & & \\
 & & & & & D & U & & & \\
 & & & & & L & D & U & & \\
 & & & & & & L & D & U & \\
 & & & & & & & L & D & U \\
 & & & & & & & & L & D
\end{pmatrix}. \tag{3.2}$$

Similarly, the forward Gauss-Seidel preconditioner is given by

$$\mathbf{P}_{\text{FGS}} = \begin{pmatrix}
D & U & & & & & & & & \\
L & D & U & & & & & & & \\
 & L & D & U & & & & & & \\
 & & L & D & U & & & & & \\
 & & & L & D & & & & & \\
 & & & & \boldsymbol{L} & D & U & & & \\
 & & & & & L & D & U & & \\
 & & & & & & L & D & U & \\
 & & & & & & & L & D & U \\
 & & & & & & & & L & D
\end{pmatrix}, \tag{3.3}$$

where we have emphasized the "tying" block (in bold) on the subdiagonal, and the backward Gauss-Seidel preconditioner is given by

$$\mathbf{P}_{\text{BGS}} = \begin{pmatrix}
D & U & & & & & & & & \\
L & D & U & & & & & & & \\
 & L & D & U & & & & & & \\
 & & L & D & U & & & & & \\
 & & & L & D & \boldsymbol{U} & & & & \\
 & & & & & D & U & & & \\
 & & & & & L & D & U & & \\
 & & & & & & L & D & U & \\
 & & & & & & & L & D & U \\
 & & & & & & & & L & D
\end{pmatrix}, \tag{3.4}$$

where we have emphasized the tying block (in bold) on the superdiagonal. The Jacobi preconditioner parallelizes trivially, as each of the blocks corresponding to the time subdomains

can be inverted independently, while the Gauss-Seidel preconditioners do not. Nonetheless, it is possible to parallelize the application of Gauss-Seidel preconditioners through a so-called "red-black" reordering, which we will investigate in a future publication.

4. Numerical results. This section examines the performance of the three preconditioners for the optimal control problem given in Section 2. We present four experiments. The first experiment is designed to compare the effectiveness of our preconditioners with those investigated in [5]. The second experiment studies the potential for their efficient use in inexact SQP methods, such as [6]. The third experiment examines preconditioner performance for a fixed size of time subdomains. The fourth experiment examines the smoothing properties of our preconditioners, with the outlook for use in new multigrid in time methods for PDE-constrained optimization.

REMARK 4.1. *For the given data* $z(t) = r(t) = 2\pi(1 - e^{-t})$, $u_0 = 0$, *and* $f(x,t) = (4\pi^2(1 - e^{-t}) + e^{-t})\sin(2\pi x)$, *the solution of the constraint equation* (2.2) *is* $u(t,x) = \sin(2\pi x)(1 - e^{-t})$. *We verify our finite element implementation of the constraint equation by using this manufactured solution, as follows. If we consider a sufficiently fine time discretization, we observe second order convergence of the numerical PDE solution in space, and if we consider a sufficiently fine spatial discretization, we get second order convergence in time for the Crank-Nicolson method* ($\theta = 1/2$). *The implementation of the optimal control problem is verified using manufactured solutions as well, where we have studied convergence of the optimal state* u_* *in space and time.*

In all cases below, $f(t,x) = (4\pi^2(1-e^{-t})+e^{-t})\sin(2\pi x)$, $r(t) = 2\pi(1-e^{-t})$, $u_0(x) = 0$, $\alpha_2 = 0$, $s_1 = 1$, and the initial guess for the iterative solution of the optimality system is zero. We use the Crank-Nicolson scheme for the time discretization.

Experiment 1. Here we compare the performance of our preconditioners with similar preconditioners investigated in [5]. We consider the penalty parameter $\alpha_1 = 10^3$. We use $N_x = 10$ spatial intervals and $N = 30$ time intervals. The GMRES method is applied with the 2-norm residual tolerance of 10^{-7}. The maximum number of iterations for GMRES is set to 100.

The iteration numbers in Figure 4.1 for the forward and backward Gauss-Seidel preconditioners are very close to those published in [5, Fig. 4]. This implies that our Gauss-Seidel preconditioners could be used for storage reduction, similar to [5]. Unfortunately, their application, as implemented here, does not parallelize in time. In contrast, the application of the Jacobi preconditioner, which was not studied in [5], parallelizes in time.

Experiment 2. Here we demonstrate the potential effectiveness of the Jacobi scheme as a time-parallel preconditioner for linear systems arising in, e.g., inexact sequential quadratic programming (SQP) methods. The key to efficiency in these algorithms is the use of very coarse, i.e., inexact, iterative solves for the optimality systems. It is reported in [7, 6] that residual tolerances as large as 0.5 can be handled robustly and efficiently by inexact SQP schemes, and that tolerance levels of 10^{-2} or 10^{-3} are optimal for many applications. We let $N_x = 100$ and $N = 100$, set the maximum number of GMRES iterations to 200, and keep all other parameters the same as in Experiment 1. While we are particularly interested in the performance of the Jacobi scheme, we report forward Gauss-Seidel results as well, see Figure 4.2.

Table 4.1 shows the GMRES iterations at which the relative residual drops below 10^{-3} and 10^{-2}, respectively, for both preconditioners. For the Jacobi preconditioner, the potential speedup due to time parallelization can be defined as the ratio of the number of time subdomains and the iteration numbers reported in Table 4.1. In the conservative case where the relative residual is 10^{-3}, the potential parallel speedup for 50 time subdomains is $50/18 = 2.8$, while the speedup for the relative residual of 10^{-2} is $50/7 = 7.1$. These

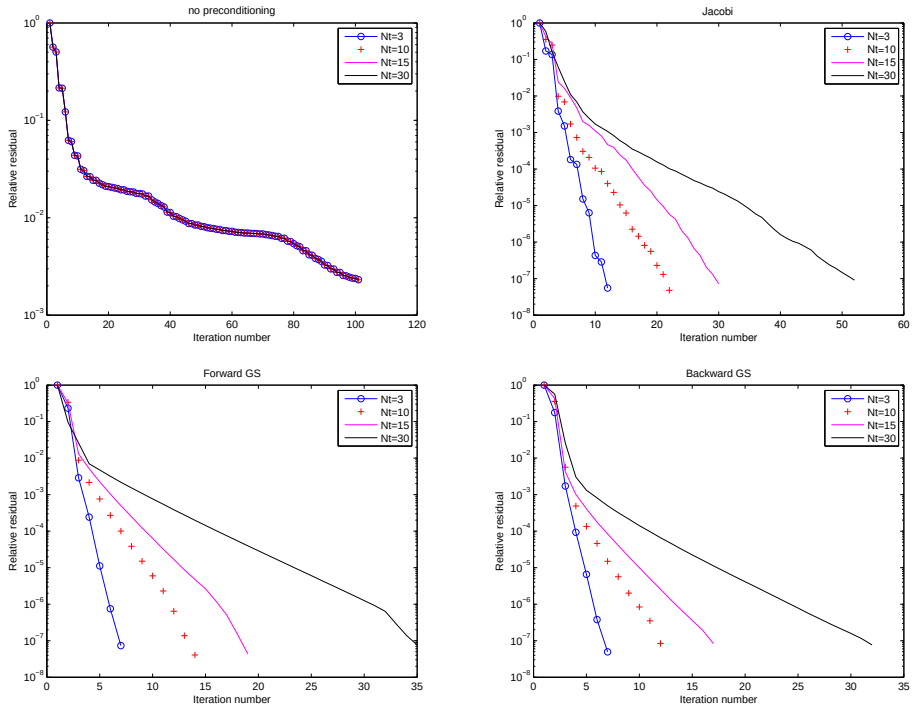


Fig. 4.1: Convergence history of GMRES with Gauss-Seidel and Jacobi preconditioners for $N_x = 10$, $N = 30$ and different numbers of time subdomains, N_t. The iteration numbers for the forward and backward Gauss-Seidel preconditioners are very close to those published in [5, Fig. 4].

results encourage a closer study of the time-domain decomposition Jacobi preconditioner in the context of inexact SQP methods.

N_t	5	10	20	50	5	10	20	50
Forward GS	4	5	8	9	3	3	3	3
Jacobi	6	8	10	18	4	4	5	7

Table 4.1: The GMRES iteration number at which the relative residual drops below 10^{-3} (left part of the table) or 10^{-2} (right part of the table), for $N_x = 100$, $N = 100$ and different numbers of time subdomains, N_t. For the relative residual of 10^{-3}, the maximum parallel speedup is $50/18 = 2.8$, while the speedup for the relative residual of 10^{-2} is $50/7 = 7.1$.

Experiment 3. Here we take $N_x = 10$, set the maximum number of GMRES iterations to 100, set the relative residual tolerance to 10^{-7}, and fix the number of time steps per time subdomain to five, $N/N_t = 5$. We vary the total number of time steps from 25 to 200. The performance of the Jacobi and forward Gauss-Seidel preconditioners is illustrated in Figure 4.3. The backward Gauss-Seidel preconditioner performs similarly to the forward Gauss-Seidel preconditioner, so we omit those results. Figure 4.3 demonstrates good iteration scaling for both preconditioners as the total number of time steps increases. Going from

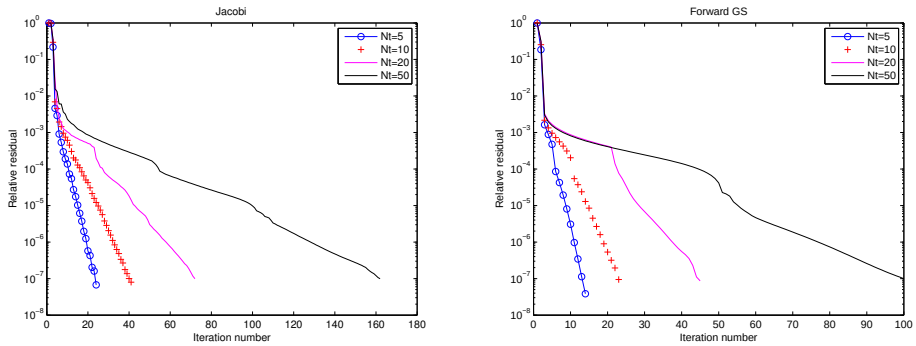


Fig. 4.2: Convergence history of GMRES with the Jacobi and forward Gauss-Seidel preconditioners for $N_x = 100$, $N = 100$ and different numbers of time subdomains, N_t. The results are analyzed in Table 4.1.

25 to 200 total time steps, the number of iterations for the Jacobi preconditioner increases from 16 to only 21, while the iteration increase for the forward Gauss-Seidel scheme is only one, from 10 to 11.

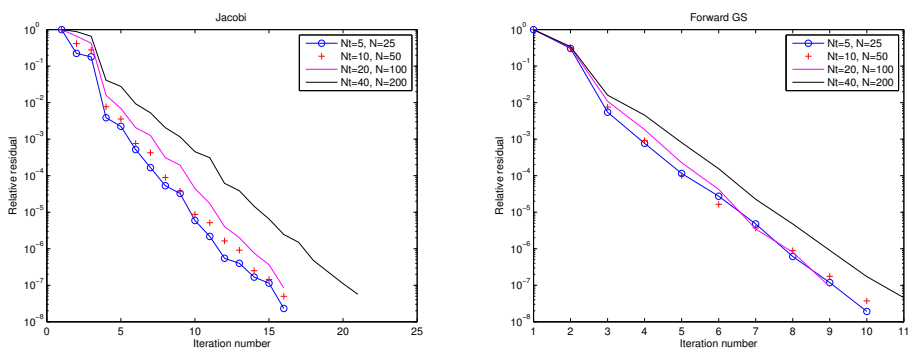


Fig. 4.3: Convergence history of GMRES with the Jacobi and forward Gauss-Seidel preconditioners for $N_x = 10$, $N/N_t = 5$, and different total numbers of time steps, N. Both preconditioners show good iteration scaling as the total number of time steps increases.

Experiment 4. Here we present an empirical study of the the smoothing properties of our preconditioners, based on error reduction, see e.g., [2]. The goal is to assess their potential as smoothers for multigrid-in-time methods. By writing $\mathbf{A} = \mathbf{A} + \mathbf{P} - \mathbf{P}$, we obtain a new form of $\mathbf{Ax} = \mathbf{b}$,

$$\mathbf{Px} = (\mathbf{P} - \mathbf{A})\mathbf{x} + \mathbf{b}, \tag{4.1}$$

where $\mathbf{P}$ could be any of the above preconditioners. This form suggests a fixed-point iteration,

$$\mathbf{Px}_{k+1} = (\mathbf{P} - \mathbf{A})\mathbf{x}_k + \mathbf{b}. \tag{4.2}$$

We subtract equation (4.2) from (4.1) to find the error equation

$$\mathbf{P}(\mathbf{x} - \mathbf{x}_{k+1}) = (\mathbf{P} - \mathbf{A})(\mathbf{x} - \mathbf{x}_k). \tag{4.3}$$

We introduce error vectors $\mathbf{e}_k = \mathbf{x} - \mathbf{x}_k$ and $\mathbf{e}_{k+1} = \mathbf{x} - \mathbf{x}_{k+1}$, multiply both sides of the equation by the inverse of $\mathbf{P}$, and arrive at the error update equation

$$\mathbf{e}_{k+1} = (\mathbf{I} - \mathbf{P}^{-1}\mathbf{A})\mathbf{e}_k. \tag{4.4}$$

This fixed-point iteration converges if the spectral radius of the operator $\mathbf{I} - \mathbf{P}^{-1}\mathbf{A}$ is less than one. We do not expect this to hold in the optimal control context, however it is informative to examine the error based on equation (4.4).

For all studies below we choose $N = 100$ and $N_x = 100$, and set the number of time subdomains to $N_t = 10$. Additionally, instead of enforcing $\bar{u}_i = u_i$ as the continuity condition, we will use $C\bar{u}_i = Cu_i$ to improve the scaling of the Lagrange multipliers. The starting condition $\mathbf{e}_0$ is a vector with each component randomly drawn from a uniform distribution over the open interval $(-1, 1)$. The goal is to seed the initial error with a range of spectral content in both space and time. Then after applying the relaxation method the resulting value empirically represents the smooth error. In the context of multilevel methods, this smooth error would be handled with a coarse grid correction. Multilevel schemes will be studied in a future publication.

We study the state and adjoint components of the error. Figure 4.4 has four plots of the state error (left column) and four plots of the adjoint error (right column). The first row shows the random initial condition. The second row gives a view of the error with respect to time after five applications of the Jacobi smoothing operator. The third row shows a colormap of the error in time and space after five iterations. The final row shows a similar colormap after 100 iterations. Figures 4.5 and 4.6 contain similar sets of plots for forward and backward Gauss-Seidel, respectively. There are two interesting features. First is the scaling of the error, and its reduction with increasing iteration counts. All methods reduce the error by at least an order of magnitude after 100 iterations. The second feature is the spectral content of the error. For all the methods, after five iterations some of the noise of the initial condition remains in the spatial direction. Also, the structure of the discontinuous-in-time solution becomes more apparent, as the preconditioners do not enforce continuity explicitly. After 100 iterations the noise in the spatial direction is completely eliminated. On the other hand, the error at the time discontinuities remains relatively large. This large error represents a "smooth mode" that can be removed using a coarse grid correction in multigrid methods.

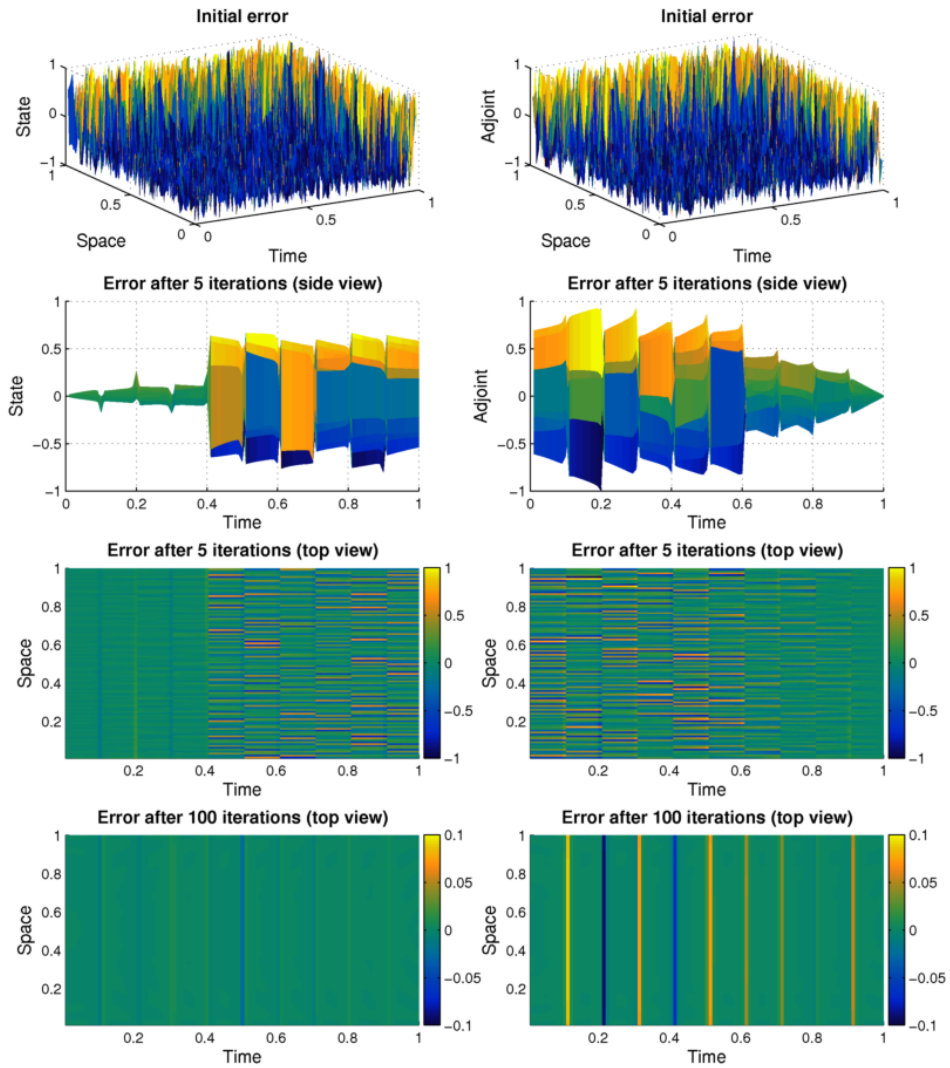


Fig. 4.4: The state component of the error (left) and the adjoint component of the error (right) for the Jacobi preconditioner at different iterations. Note the difference in scales between the top six images and the bottom two images.

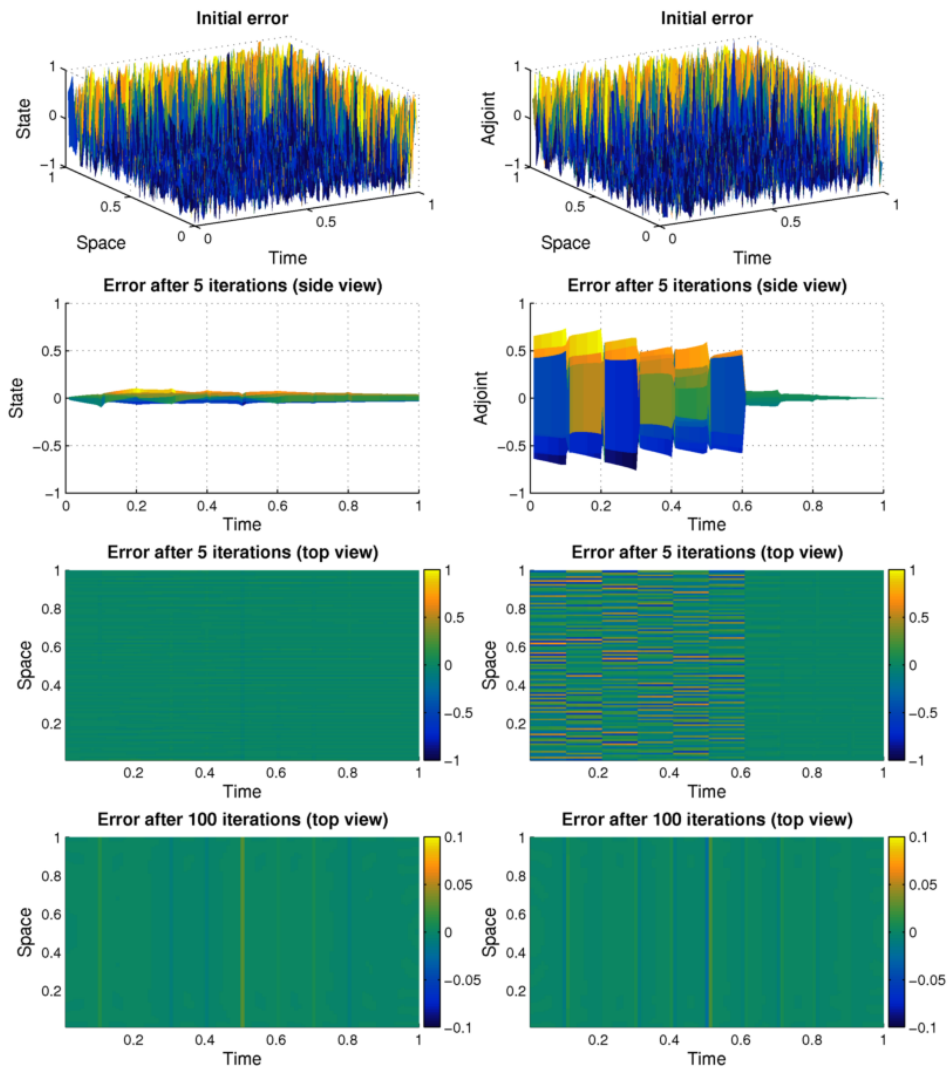


Fig. 4.5: The state component of the error (left) and the adjoint component of the error (right) for the forward Gauss-Seidel preconditioner at different iterations. Note the difference in scales between the top six images and the bottom two images.

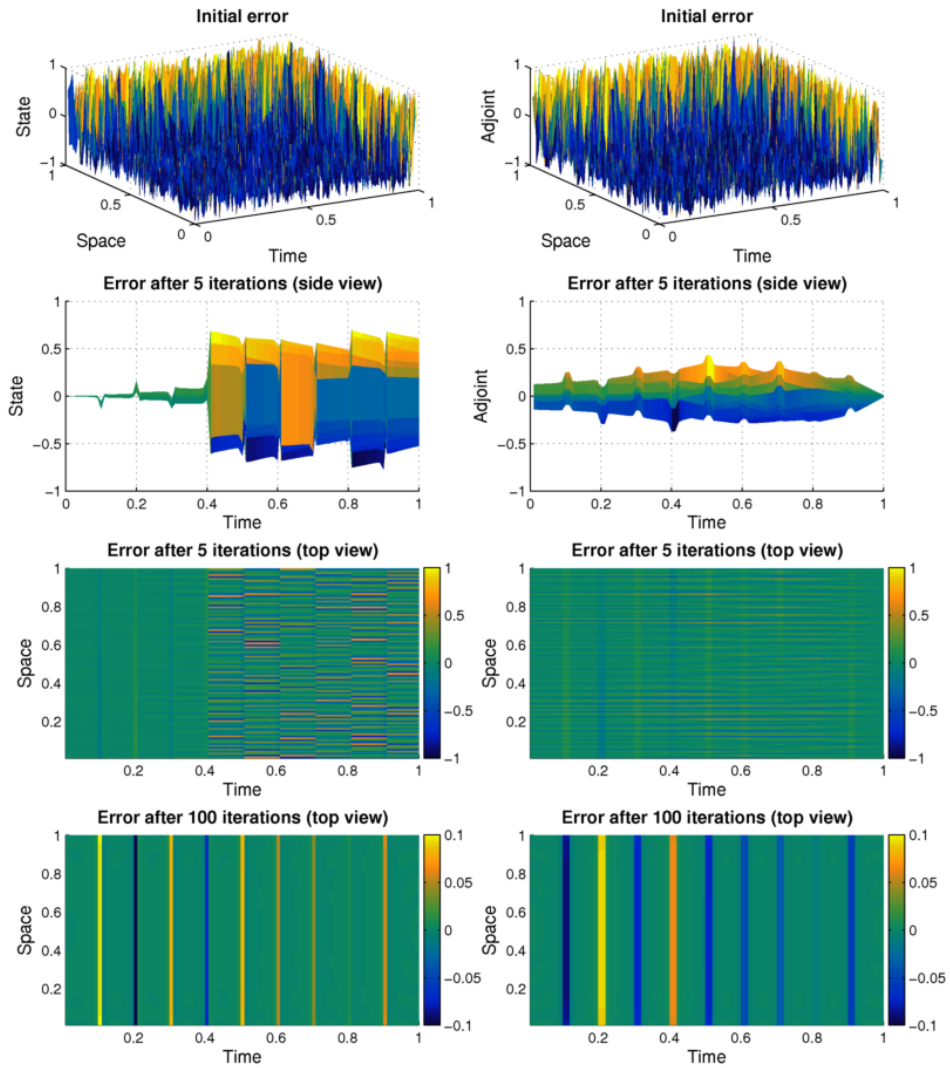


Fig. 4.6: The state component of the error (left) and the adjoint component of the error (right) for the backward Gauss-Seidel preconditioner at different iterations. Note the difference in scales between the top six images and the bottom two images.

5. Conclusions and future work. We studied a time-parallel approach to solving quadratic optimization problems with linear time-dependent PDE constraints. Our approach is based on time-domain decomposition, block tridiagonalization and the subsequent use of fixed-point iterations with matrix splittings. We demonstrated the potential effectiveness of the Jacobi scheme as a time-parallel preconditioner for linear systems arising in inexact SQP methods. We also showed that the Jacobi and Gauss-Seidel schemes may be good smoothers for multigrid in time methods, which we will study next. We will also integrate the proposed schemes into inexact matrix-free SQP methods for nonlinear optimization, and tackle optimization problems with nonlinear time-dependent PDE constraints.

REFERENCES

[1] J. Albersmeyer and M. Diehl, *The lifted Newton method and its application in optimization*, SIAM J. Optim., 20 (2010), pp. 1655–1684.

[2] W. L. Briggs, S. F. McCormick, et al., *A multigrid tutorial*, SIAM, 2000.

[3] A. Comas, *Time-domain decomposition preconditioners for the solution of discretized parabolic optimal control problems*, ProQuest LLC, Ann Arbor, MI, 2006. Thesis (Ph.D.)–Rice University.

[4] M. J. Gander, *50 years of time parallel time integration*, in Multiple Shooting and Time Domain Decomposition, Springer, 2015. In press.

[5] M. Heinkenschloss, *A time-domain decomposition iterative method for the solution of distributed linear quadratic optimal control problems*, J. Comput. Appl. Math., 173 (2005), pp. 169–198.

[6] M. Heinkenschloss and D. Ridzal, *A matrix-free trust-region SQP method for equality constrained optimization*, SIAM J. Optim., 24 (2014), pp. 1507–1541.

[7] D. Ridzal, M. Aguiló, and M. Heinkenschloss, *Numerical study of a matrix-free trust-region SQP method for equality constrained optimization*, Tech. Rep. SAND2011-9346, Sandia National Laboratories, 2011.

[8] Y. Saad and M. H. Schultz, *GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems*, SIAM J. Sci. Statist. Comput., 7 (1986), pp. 856–869.

EXPLOITING DOMAIN KNOWLEDGE TO OPTIMIZE MULTI-SCALE PERIDYNAMICS COMPUTATIONS

MUHAMMAD HASAN JAMAL* AND DANIEL Z. TURNER†

Abstract. The development of *multi-scale computational methods* is rapidly increasing which enable complex problems to be solved at multiple spatial and temporal scales. These problems are usually solved by decomposing a large problem domain into multiple sub-domains that are solved independently at different timescales and granularity and then coupled back to get the desired solution. However, a particular problem can be solved in numerous ways by coupling the sub-problems together in different orders. Our previous work [10] showed that the search space and performance variance of possible coupling orders is large, and finding the best order to solve a problem is non-trivial. We presented a system that uses recursive decomposition to generate efficient coupling schedules automatically by exploiting domain semantics and domain-specific cost models for local multi-scale computational mechanics methods. The goal of this paper is to validate that our scheduling approach can be applied to any multi-scale problem domain by just replacing the domain-specific cost model for that domain. In this paper, we apply our scheduling approach on non-local multi-scale computational peridynamics method. We show that our cost model is highly correlated with actual application runtime. The coupling schedule generated by our approach, when executed sequentially or in parallel using the parallel run-time system (Cilk) [3], outperforms alternate scheduling strategies, as well as over 99% of randomly-generated coupling schedules sampled from the space of possible solutions.

1. Introduction. Multi-scale methods have seen significant development in recent years for problems in science and engineering [4, 5, 11]. These methods allow problems domains to be decomposed into smaller parts and simulated with vastly different spatial and temporal scales in the different parts. This allows fine grain simulation for areas of interest, at higher computational cost, while the remaining parts of the problem can be approximated with a much coarser model. Multi-scale methods thus avoid incurring the cost of such fine-grained simulation throughout the problem domain while still assuring necessary level of resolution for simulation.

Figure 1.1 shows an example of a crack growth and propagation. Such systems can be solved at multiple scales where the area around the crack that needs to be investigated at a finer granularity is run at a smaller timescale while the rest of the system is run at a larger timescale.

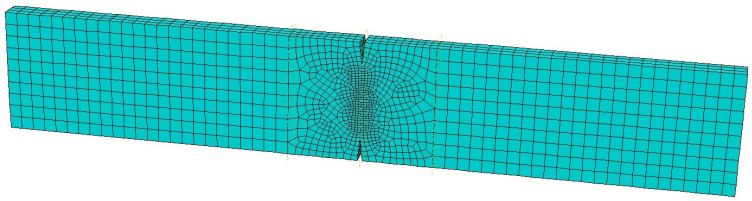


Fig. 1.1: Beam Mesh with a Notch

Recursive domain decomposition [1, 2, 6, 8, 13, 18] is an attractive approach for multi-scale simulation where a large problem is decomposed into a group of loosely-coupled sub-domains. These sub-domains are solved independently, except at shared *interfaces* that connect them. To ensure consistency of solution at these interfaces, the solutions of each individual sub-domain are required to be coupled at these shared interfaces. It is computa-

*School for Electical and Computer Engineering, Purdue University, jamal0@purdue.edu
†Sandia National Laboratories, dzturne@sandia.gov

tionally advantageous to decompose a large system instead of solving it as a single entity as long as the size of interfaces are relatively small compared to the size of the sub-domains.

Figure 1.2 shows an example of this approach that results in a tree-like solution algorithm. A square problem domain with a crack is decomposed into four sub-domains such that the sub-domains adjoining the crack are simulated at a smaller timescale than the other two, as the area around the crack is "the area of interest". For each larger ΔT in the system, the sub-domains at smaller Δt are solved $\boldsymbol{m}$ times, where $\boldsymbol{m}$ is the integer ratio between ΔT and Δt. After solving for each sub-domain, individual sub-domain solutions are then coupled together, in a recursive manner, to obtain the final solution. Once the interface between two sub-domains is solved at each step, these sub-domains are considered as a single large sub-domain in the next step and is represented by an interior node in the tree. After the final coupling operation at the root node, the final solution is obtained.

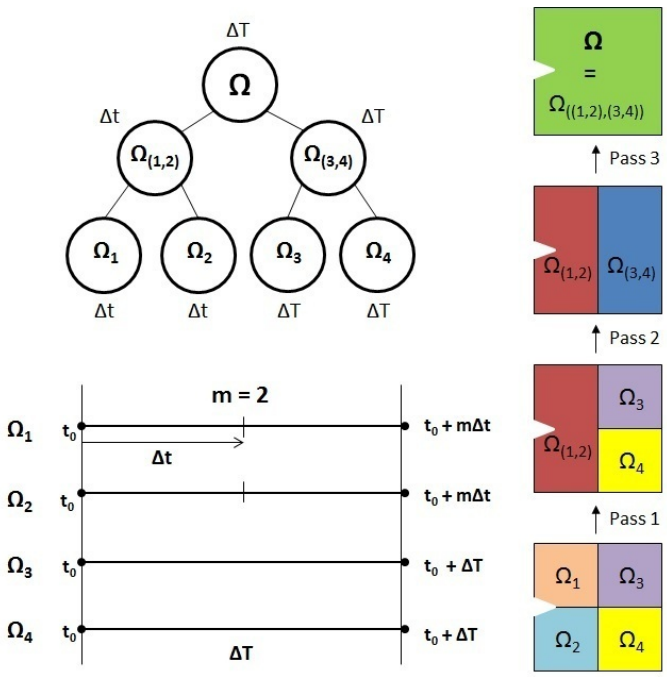


Fig. 1.2: Recursive domain decomposition and coupling for multi-scale problems

As we show in [10], the critical point about this hierarchical approach to solving multi-scale problems is that the *structure* and *topology* of the coupling tree has a significant effect on the performance of the algorithm. Because the coupling operation is both commutative and associative, a vast number of unique coupling trees are possible (945 for a problem with just 6 sub-domains), making it highly unlikely for all but the simplest problems that a domain scientist will adopt an effective coupling order. Furthermore, because the various relevant parameters associated with computational costs are problem-dependent, finding the optimal coupling tree becomes even more difficult, as no single approach for finding an optimal tree coupling order may work for all different problems.

To tackle this problem, in our previous work [10], we developed a run-time optimization system based on the inspector-executor approach that first inspect the original program to infer the high level computational structure of the problem and then uses semantic properties

of the coupling operation, as well as a domain-specific cost model to obtain cost values that reflect the coupling of multi-scale sub-domains. Next an optimal coupling order is generated based on those cost values. Finally, the optimized schedule is run using the Cilk framework to exploit available parallelism.

In this paper, from a peridynamics model, we infer problem semantics and use a domain-specific cost model along with our scheduler from our previous work to optimize a non-local bond-based computational peridynamics method.

Finally, we discuss that with our overall generalized approach, how computational algorithms and domain applications from other domains can also be optimized using similar semantics-exploiting approaches without much effort from domain scientists in optimizing their code.

2. Background.

2.1. Peridynamics. Peridynamics is a non-local solid mechanics theory that permits the formation of discontinuities in the displacement field, such as cracks and fractures. In contrast to the classical local theory and also most other non-local approaches, the peridynamic equation of motion is free of any spatial derivatives of displacement. It replaces the partial differential equations (PDEs) of classical continuum theories with integro-differential equations (IDEs). A material point x in a body interact directly with other material points within a distance δ called the horizon. The material within a distance δ of x is called the family of x, H_x. In 3D the horizon will be a sphere. Figure 2.1 provides an overview of a material point and its interactions. The interaction between material point x and any material point q in its family is called a bond which is a pairwise force density vector f(q,x,t) that is applied both material point. This pairwise bond force vector is jointly determined by the collective deformation of H_x and H_q. These bond forces are anti-symmetric, i.e. f(q,x,t) = -f(q,x,t). Figure 2.2 shows two material points in an undeformed and deformed states. y is the deformation. The concept of a bond that extends over a finite distance is a fundamental difference between the peridynamic model and classical models for materials which are based on the idea of contact forces, i.e., interactions between material points that are in direct contact with each other.

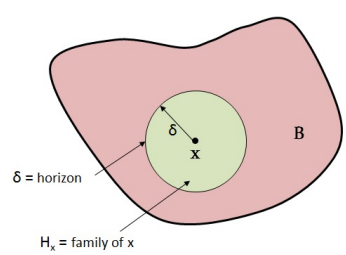


Fig. 2.1: An overview of an interaction of a material point in peridynamics

Boby forces are applied over a volumetric region. These forces can be constant or time-variant, and are representative of the body forces on the material. Damage is calculated by comparing the current stretch, given by the ratio of the relative deformation between two nodes to their undeformed relative position, to some critical stretch S_0 . The stretch must be checked for every bond formed with every pair of material points. If any bond stretches beyond the critical limit, that bond is broken and its force contribution is negligible in future calculations.

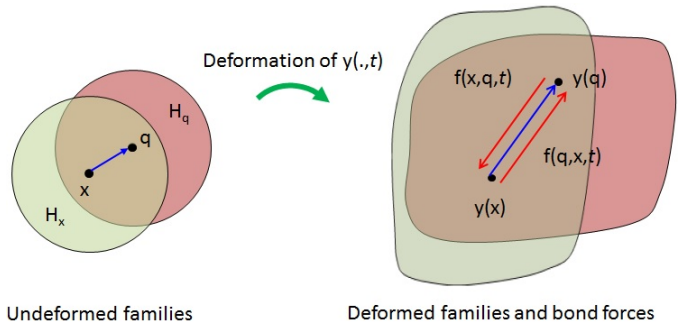


Fig. 2.2: Deformation in peridynamics

To solve the peridynamics IDEs system, the system is spatially discretized into volume sections, where each section is represented by a node in a peridynamic grid. The discretization process is explained in [19]. This system is then temporally discretized and solved by finite-difference schemes by approximating the time derivatives using difference formulas. This allows one to solve for the state $\boldsymbol{u}_{n+1} \approx \boldsymbol{u}(t_{n+1})$ of the system at some time t_{n+1} from a known state $\boldsymbol{u}_n \approx \boldsymbol{u}(t_n)$ by advancing through a small timestep Δt where $t_{n+1} = t_n + \Delta t$. This process can be repeated successively to advance the solution in time [12]. However, depending on the problem, the size of the this system of equations can be very large making its solution as a single complete system very computationally intensive. One way to avoid solving this system as a whole is by using domain decomposition. This approach divides the problem into smaller sub-domains, solves them independently (possibly in parallel), and couples them back by enforcing continuity constraints on the interfaces between the sub-domains.

2.2. Solving a Recursively Decomposed Domain. Given a peridynamic grid of nodes, partitioned into $|S|$ sub-domains where $S = \{A, B, ...\}$, a hierarchy of sub-domains can be built by combining two sub-domains at a time until the original undecomposed grid is recreated as shown in the *tree* structure, in Figure 1.2. The original undecomposed structure Ω is represented by the *root* node and the *leaf* nodes represent sub-domains that are not further subdivided.

In general, the problem of solving any node $\Omega_{(A,B)}$ in the tree, is substituted with two coupled sub-problems for Ω_A and Ω_B. The entire system can be solved in a decomposed manner, using the following sequence of operations at each timestep:

1. Solve the smaller uncoupled sub-domain problems for $S = A, B$.
2. Solve for the interface variable $\boldsymbol{\lambda}$.
3. Update the individual solutions for Ω_A and Ω_B.

We refer to the above solution procedure as $\mathsf{TreeSolve}(\Omega_{(A,B)})$. Calling $\mathsf{TreeSolve}(\Omega)$ at the root node and recursively solving the entire tree advances the state of the original problem from $\boldsymbol{u}_n$ to $\boldsymbol{u}_{n+1}$.

It has been shown that the recursive domain decomposition method allows for any decomposition of a given mesh into sub-domains and also allows the sub-domains to be coupled back in any order to yield the original problem domain. This fact results in a very large number of possible tree representations of the coupled solution procedure for a given set of sub-domains. For instance a problem decomposed into two sub-domains can be coupled in only one way resulting in a unique tree. However, 3 sub-domains can be coupled in 3 different ways and correspond to 3 distinct trees, 4 sub-domains result in 15 distinct

trees, 8 sub-domains yield 135,135 trees, 16 sub-domains have about 6.19×10^{15} trees, and so on. As we increase the number of sub-domains, the number of possible trees becomes enormously large. Picking a particular tree that results in good performance out of this huge space of possible trees is not trivial.

2.3. Multiple Time Scales. In section 2.2 we discussed a method that utilizes a single timescale for discretizing all the different sub-domains in a tree. Usually in large systems, the area of special interest is small (e.g. crack in a beam) which needs to be simulated at a finer granularity for numerical stability and accuracy, compared to the rest of the system. Solving these types of problems using a single, very small, timescale (that is governed by the fastest dynamics around the crack) is impractical and computationally unnecessary since we only want high fidelity in some parts of the problem domain. To overcome this problem, such problems are solved at multiple timescales, smaller (finer) timescales for the sub-domains in and around the area of interest and larger (coarser) timescales for rest of the sub-domains.

The approach described in section 2.2 is most beneficial when different sub-domains are solved at different timescales and can be modified to account for such cases. In step (1) of the TreeSolve procedure, if the daughter nodes Ω_A and Ω_B are assigned different timescales Δt_A and Δt_B with an integer timescale ratio $m = \Delta t_A/\Delta t_B$ between them, then sub-domain Ω_B is simply solved m times for every time that sub-domain Ω_A is solved. As an example, consider the same beam problem in Figure 1.2 with a timescale ratio of $m = 2$. To study the notch at a finer granularity, sub-domains Ω_1 and Ω_2 will be simulated at smaller (finer) timescale Δt while the remaining two sub-domains will be simulated at a larger (coarser) timescale ΔT. Thus, sub-domains Ω_1 and Ω_2 are solved twice for each solve of sub-domains Ω_3 and Ω_4. Coupling between the timescales in step (2) of the TreeSolve procedure, is done only once, at the larger timescale Δt_A. Finally the update in step (3) requires that the solutions for sub-domains Ω_A and Ω_B be updated in the same ratio as their timescales. The multi-timescale method is valid for any number of timescales in the problem, however, in this study we are focused on the most common scenario i.e. two different timescales. Finer details of the multi-timestepping method can be obtained from the references [16, 17].

2.3.1. Constraints. Due to the additional complexity of handling multiple timescales, there is one constraint that multi-timescale method imposes on the choice of timescales. In any tree, all child nodes are always supposed to be at a smaller (finer) timescale or the same timescale as their parent node. In addition all sub-domains at the same timescale value must be coupled prior to coupling with sub-domains with a different timescale.

Figure 1.2 shows a generic way of solving local and non-local models using domain decomposition. For peridynamics, additional restrictions are applied. To solve multiple sub-domains at different timesteps, there must be a volumetric interface region where the nodes of the sub-domains overlap of width at least equal to the horizon δ, similar to applied forces and boundary conditions. Figure 2.3 shows an initially cracked plate with loads applied at both ends. The area of interest is the crack so Ω_B will be run at a smaller timestep while Ω_A is simulated at a larger timestep. The interface between Ω_A and Ω_B is a volumetric region Ω_I where the nodes are shared between Ω_A and Ω_B. Once both sub-domains Ω_A and Ω_B are solved, the solution on Ω_A and Ω_B is coupled using Lagrange multipliers [14, 15].

For expediency, we use 1-D decomposition of the problem, hence a volumetric interface region cannot be shared by more than two sub-domains. For more information on the theory and mathematics behind multi-timesteping applied to peridynamics, see [9].

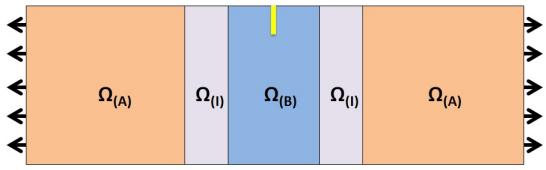


Fig. 2.3: A two sub-domain plate with an initial crack, loaded in tension

3. Cost Model and Tree Building Heuristics. In our previous work [10], we described the properties of our coupling trees, which lays the foundation for creating valid tree orderings. The coupling operations are both associative and commutative. Hence given a coupling tree, we can create new trees by tree rotation and swap operations. We showed that the number of distinct valid trees is exponentially large and that it is not an easy task to find an optimal ordering. To obtain a good solution, we introduced a cost model which correlates with the computational time taken to complete the TreeSolve routine. We used heuristics to find an optimal tree coupling order that minimizes computational cost, thus reducing execution time and gaining performance.

In this multi-scale peridynamics work, we use the same tree building heuristics as the properties of coupling operations (associativity and commutativity) in local methods and non-local methods will be same. For peridynamics, cost model will change as the computations are based on cells and bonds between cells as compared to nodes and elements of meshes in the local methods. This section discusses the cost model that produces a low cost coupling schedule automatically.

3.1. Computational Cost Modeling. As described in section 2.2, the three main operations needed to solve a given decomposed mesh are (i) uncoupled sub-domain solve, (ii) interface solve for coupling, and (iii) update. We adopt a simple cost model to quantify the cost of these operations. This is done by analyzing the input problem to determine its properties, namely the interface size ($\boldsymbol{m}$) and sub-domain sizes ($\boldsymbol{n}$). The interface size is the number of equations that need to be solved for the nodes shared between two sub-domains. The sub-domain size is the number of equations needed to be solved in an individual sub-domain. The nodes at the shared interface are also considered for individual sub-domains. With this information, we can approximate the matrix sizes for all operations to estimate the time it will take to run during actual execution, without actually running execution step. For expediency, we consider only a dense direct linear solver for our cost model, although our linear systems are not dense. It is hard to come up with a cost model for sparse solvers. The cost of sub-domain solves is of order $\boldsymbol{n}^3$, the coupling cost is of order $\boldsymbol{m}^3$, and the cost of updating the sub-domain solutions is $\boldsymbol{n_1^2 m} + \boldsymbol{n_2^2 m}$. For multi-timescale problems, we simply multiply the cost of each operation by the number of times it is executed within a single timestep.

In our results, we show that the cost values correlate well with actual runtimes, enabling us to use heuristics based on the models to create effective coupling trees.

Figure 3.1 shows CDF of tree cost for 500 randomly generated trees, for a given cuboid problem decomposed into 16 sub-domains. It is observed that by picking only 500 randomly coupled trees out of the whole tree space ($< 1\%$ of the space), the costs varies. Our work presents a way to find a configuration that obtains a near-optimal runtime among the space of coupling trees.

One point to note here is that the variation in the costs in not as significant as our previous work using local methods [10]. That is because we use 1-D decomposition for input

problem. The cuboid problem we used here, is decomposed in equal sized sub-domains and the interfaces are of similar size. With these limitations different coupling orders will have similar costs. We believe that with peridynamics methods that can handle interfaces shared between more than two sudomains, we will see significant performance variation across trees.

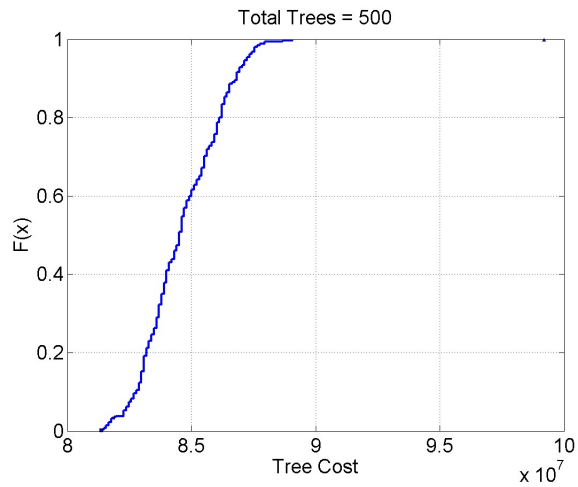


Fig. 3.1: CDF of 500 Random Trees based on tree costs

3.2. Tree Building Heuristics. With a large space of existing possible trees for a given problem, the probability of a domain scientist choosing a low cost tree is small. A simple way would be to couple sub-domains based simply on the partition labeling of the sub-domains (i.e., the order in which partitioners such as METIS [7] produce the partitions). We call this the Default approach, and it represents the baseline coupling order for a problem.

Given a cost model that computes costs on a per-subtree basis, an intuitive approach would be to develop a greedy algorithm that builds the coupling tree from the bottom up. We discuss its limitation in [10].

Instead, we adopt a top down approach, focusing on minimizing the coupling costs at higher levels of the tree. Beyond this, we would also like to maintain the balance of the tree, with both left and right subtrees for any node having roughly equal solution times; this will improve the potential parallelism of the resulting tree.

In order to facilitate tree building, the decomposed system can be abstracted as an undirected graph, as shown in Figure 3.2. In the graph, each node represents an individual sub-domain, and edges represent shared interfaces between them. In essence, the graph represents the topology of the sub-domains. The weights on graph nodes and edges represent the cost of their sub-domain solves and coupling operations respectively. Figure 3.2 is just a general abstraction. Using 1-D decomposition for input problems, the graph will be in the form of a linked list as only two sub-domains can share interface between them.

One approach to building a tree top-down given this abstraction is to simply perform recursive bisection of graph where each partition represents the sub-domains that will be in the left and right subtrees of the root. In the multi-time-scale case, we produce multiple graphs, one for each timestep in the problem (as sub-domains at each time step must be coupled together before moving on to larger time steps).

Note that even performing this top-down bisection relies on domain knowledge: because this tree-building approach can produce arbitrary results, it is only legal because we know

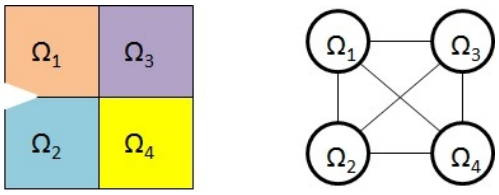


Fig. 3.2: Graph representing a 4 sub-domain system

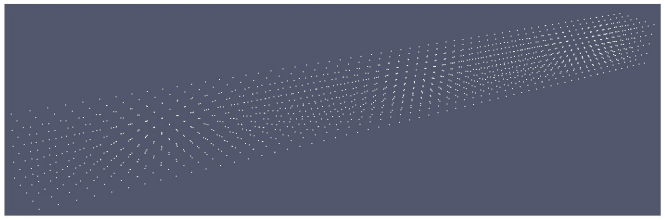
the domain semantics allow for any coupling order. Nevertheless, despite leveraging domain semantics to build the tree, this approach does not consider that leaf solve and coupling costs are based on properties of the operations such as the number of equations and interface sizes. This bisection technique is what an application programmer might think as optimal for decomposing meshes in a top-down fashion. However, this method (labeled cost-agnostic in our experiments) does not take into account domain specific cost information.

We use domain-specific heuristic from [10], which is a scheduling procedure that not only integrates domain-specific semantic information, but also domain-specific cost information. We use domain knowledge obtained from our cost model and apply the coupling and sub-domain costs to the edge and node weights, specified in section 3.1, in our graph prior to partitioning. For information on why a domain-specific heuristic works better than cost-agnostic heuristic, see [10].

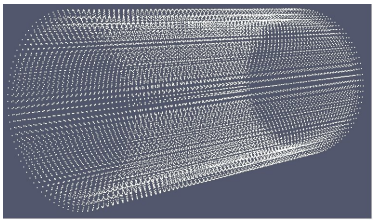
4. Evaluation. In this section, we validate our cost model against actual runtime performance and evaluate the performance of the cost-agnostic and domain-specific heuristics running both sequential and parallel TreeSolve algorithms. For our parallel implementation, we use Cilk [3] to obtain optimal parallel performance. We focus on the results of two physical testing systems: 1) a 16 sub-domain cuboid with 1500 cell (5 x 5 x 60) and 2) a 16 sub-domain cylindrical pipe with 15300 cells. Each system was partitioned by hand. These systems are simulated using the extended version of the multi-scale peridynamics solver [9] running on an Intel Xeon E5-4650 system configured with four 8-core processors (total of 32 cores) running at 2.7 GHz. Each input contains two timescales, with one fourth of the sub-domains assigned to run at a smaller timestep and the rest of the system at a larger timestep, with a timestep ratio of 2. The timestep values are arbitrarily chosen but within accuracy and numerical stability range. Figure 4.1 shows the two input systems.

4.1. Cost model validation. To validate our cost model, a set of 500 randomly generated coupling trees was used to represent a sample of the entire space of coupling trees. These trees are created by starting with all sub-domains in separate sub-trees and randomly selecting two sub-trees to be coupled together until the full tree is obtained. Note that this random coupling means that occasionally two sub-domains that do not share an interface will be coupled. This schedule is mathematically and semantically correct: the final solution will be computed correctly. Nevertheless, such coupling orders are nonsensical from a performance perspective, as there is no benefit in reduced work to be gained from coupling sub-domains that are not adjoining.

The cuboid input shown in Figure 4.1(a) with 16 sub-domains was used to explore the space of trees. Figure 4.2 illustrates the correlation of our cost model with the actual runtimes for solving the 16 sub-domain problem on single and multiple threads. For various number of cores, actual runtimes are plotted against the projected cost of the corresponding trees. The results show that the correlation between the projected cost and expected



(a) Cuboid Input



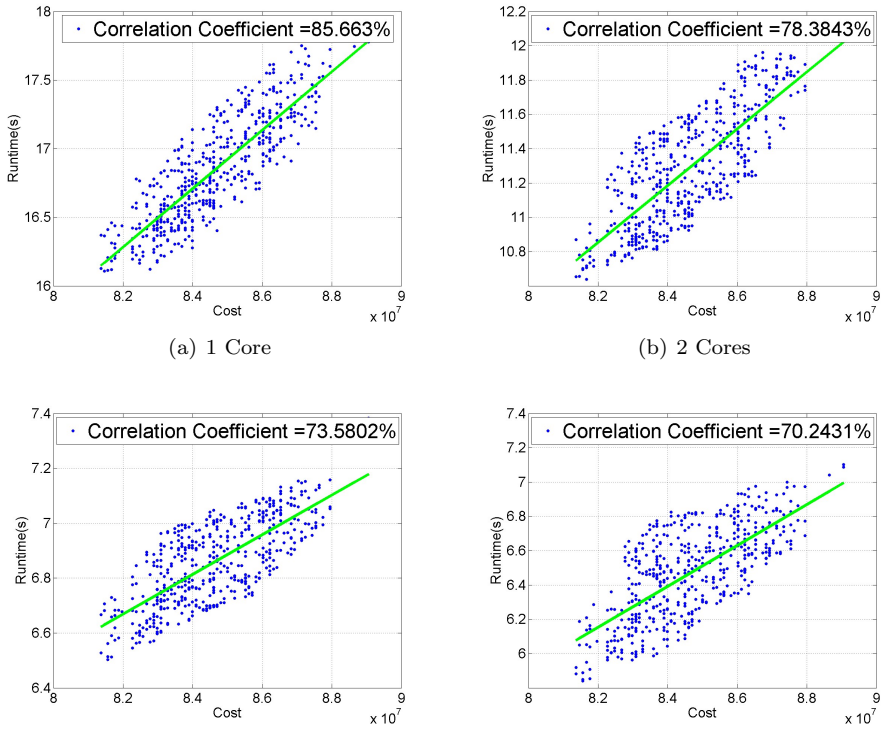
(b) Cylinder Input

Fig. 4.1: Visualization of Input Systems

runtime has an average correlation coefficient of 0.70–0.85 for single and multiple threads. We estimate the parallel costs using the same sequential model. We justify that the parallel runtime correlates with total work during parallel execution. Although we see that correlation does decrease for higher number of threads, our cost model does strongly reflect the execution time of the TreeSolve code. For the remaining test inputs, the cost model shows similar results with high correlation.

4.2. Performance comparisons. We next compare the execution times for running our heuristics on our two testing inputs. Given a partitioned input, our heuristics generate new coupling orders to solve each problem, subject to multiple timescales. In addition to our domain-specific heuristic DS, we evaluate the cost-agnostic tree CA and the Default DT, which is based on the initial partition labeling (e.g. sub-domain 1 is coupled with 2, 3 with 4 and so on). As described in Section 3.2, DT serves as a baseline. CA uses a top-down approach to produce a new tree, but does not incorporate the domain-specific cost models, while our DS heuristic refines the top-down approach by incorporating knowledge of leaf solve and coupling costs. For each input, we evaluate all three schedules. Figure 4.3 shows a CDF of the runtimes for running the 16 sub-domain cuboid input for various numbers of threads. We compare the execution times of DS, CA, and DT, along with the 500 randomly generated trees for single and multiple threads. Here we observe that DS outperforms all the trees that we tested and we achieved a significant speedup over randomly selected trees in the configuration space. In other words, despite the vast configuration space, by incorporating domain-specific semantics and cost knowledge, we are able to infer a very effective coupling order. We also note that the other evaluated schedules, DT and CA perform worse than our domain-specific schedule. We see that the default schedule is quite slow, while CA yields better results. Nevertheless, our DS approach outperforms the CA schedule by 2 to 7%.

Observations. The test input files used for this study were partitioned in one direction such that the size of all sub-domains is equal and the size of each interface between two node is approximately equal. Usually, when an input problem is decomposed with partitioners, the labeling of sub-domains might not be straight forward. To mimic this behavior, we



(a) 1 Core (b) 2 Cores

(c) 4 Cores. (d) 8 Cores.

Fig. 4.2: Cuboid: Cost vs Runtime Correlation of 500 Random Trees

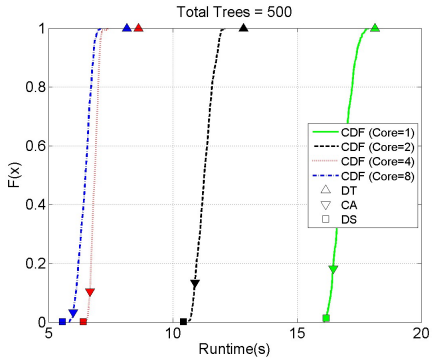
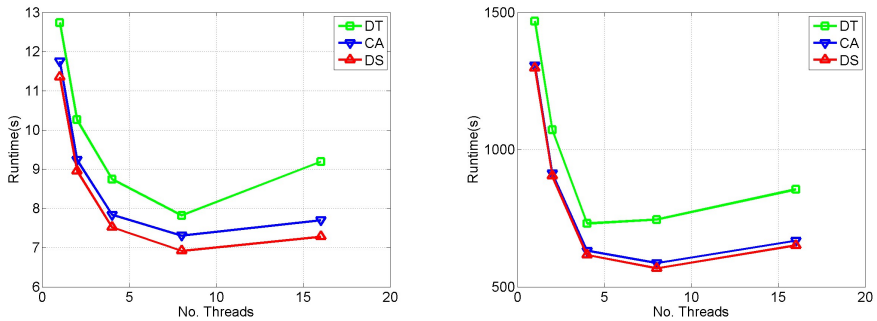


Fig. 4.3: Cuboid: CDF of Runtimes for 500 Random Trees with Heuristics

randomly assigned ID labels to sub-domains. The reason for poor performance of DT is this random labeling since DT is dependent on sub-domain labeling. We also tested the inputs by labeling then in a sequence (e.g. 1, 2, 3 ...) which is not shown in the results. This sequence labeling with DT produced exactly the same results as DS. That is understandable since the input files were nicely partitioned into equal sized sub-domains with equal sized

interfaces. DT being dependent on the order of labeling will give varied results with different labeling sequences. Our DS is independent of labeling and will generate optimized results. In the worst case, where DT can figure out the optimized schedule, DS will perform as good as, if not better than DT.



(a) 16 Sub-domain Cuboid Input (b) 16 Sub-domain Cylinder Input

Fig. 4.4: Execution times across different number of threads

4.3. Parallel performance. Figure 4.4 compares the execution times for each heuristic across different number of threads for the cuboid and cylinder inputs. We note that in all cases, the DS approach delivers better performance than the CA and DT. While the specific amount of improvement when using our DS heuristic is problem-dependent, we see that incorporating domain knowledge provides a consistent edge across multiple inputs. Table 4.1 shows the speedup of CA and DS over the baseline, DT, for each of the inputs at both one and eight threads. Not only does the DS schedule perform the best in single-threaded execution, the advantage increases at higher thread counts; DS provides better scalability. DS only slightly performs better than CA because of the nature of the input problem. We believe that DS will perform significantly better than CA for input problems with unequal sub-domain and interface sizes. DS will show significant performance improvement on peridynamics methods that allow more than two sub-domains to share a single interface. For all

Table 4.1: Speedups of CA and DS schedules over baseline DT

	No. Thread = 1		No. Thread = 8	
	CA	DS	CA	DS
Cuboid	1.08	1.12	1.07	1.13
Cylinder	1.12	1.13	1.27	1.31

of the schedules, performance scales up to 4 threads and speedup is obtained when increasing the number of threads. For 8 threads and beyond, the results do not scale well. Using the Cilk profiling tool, we found that for each input, past 4 cores, the runtime of critical path equals the runtime of the solver. This indicates that the parallelism saturates at 8 threads and not much more parallelism can be exploited. Increasing communication costs and parallel overhead plays a role here. This lack of scalability is understandable because the structure of the input is inherently imbalanced (4 sub-domains at smaller timestep vs. 12 sub-domains at larger timestep). In other words, there is a certain amount of inevitable,

application-specific load imbalance, especially when there are only a few sub-domains run at small time scales. Additionally, the tree-based nature of the algorithm and the fact that coupling occurs atomically limit available parallelism.

Available parallelism and scalability across heuristics. A natural question, given the inherent lack of scalability, is whether our scheduling heuristics negatively impact the available parallelism of the program. To investigate this question, we used Cilk to profile the total work, critical path length (span), and amount of parallelism (work/span) afforded by each schedule across all of the inputs. Table 4.2 summarizes these results.

There are two key take-aways from these results. First, as expected, the DS scheduling heuristic, which incorporates domain-specific semantic and cost information, yields the best single-threaded performance (work). This is consistent with the raw performance numbers shown above. Second, despite the heuristics focusing on work minimization, with parallelism only of secondary importance, parallelism is not adversely affected. In fact, on both inputs DS exhibits the same amount of parallelism as CA.

Table 4.2: Inherent parallelism in coupling schedules

Input	Schedule	Work (sec)	Critical Path (sec)	Parallelism
Cuboid	DT	12.74	8.12	1.57
	CA	11.76	6.57	1.79
	DS	11.35	6.40	1.77
Cylinder	DT	1468.51	789.82	1.86
	CA	1307.09	619.57	2.11
	DS	1297.73	613.92	2.11

5. Conclusions. This paper utilizes the effective scheduling algorithm from previous work [10] to optimize multi-scale computational peridynamics codes based on recursive domain decomposition. Solving a decomposed problem is represented as solving a binary tree, and the structure of the tree dictates the performance of the solver. We demonstrate for real problems that our scheduling algorithm chooses an optimized tree coupling from an exponentially large search space automatically, where there is a large variation in performance between each tree in the search space. Our scheduling algorithm consistently produce trees that rank in the 99th percentile of possible trees for problems. Finally, we show that execution of the trees from our scheduling algorithm deliver scalable performance on multiple cores as long as there is parallelism to be exploited, providing solutions in less time than randomly selected coupling trees.

Our work in this paper and previous work [10] automatically provides optimized, parallel implementations of multi-scale computational methods. We validate that our scheduling approach can be applied to any multi-scale problem domain by just replacing the domain-specific cost model for that domain, allowing domain scientists to take advantage of novel computational techniques without devoting substantial time to the tedious process of optimizing a parallel implementation on a problem-by-problem basis.

REFERENCES

[1] P. Arbenz, U. L. Hetmaniuk, R. B. Lehoucq, and R. S. Tuminaro, *A comparison of eigensolvers for large-scale 3D modal analysis using AMG-preconditioned iterative methods*, International Journal for Numerical Methods in Engineering, 64 (2005), pp. 204–236.

[2] J. K. Bennighof and R. B. Lehoucq, *An automated multilevel substructuring method for eigenspace computation in linear elastodynamics*, SIAM Journal on Scientific Computing, 25 (2004), pp. 2084–2106.

[3] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, *Cilk: An efficient multithreaded runtime system*, in Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP '95, 1995, pp. 207–216.

[4] J. Fish, *Bridging the scales in nano engineering and science*, Journal of Nanoparticle Research, 8 (2006), pp. 577–594.

[5] V. Gravemeier, S. Lenz, and W. A. Wall, *Towards a taxonomy for multiscale methods in computational mechanics: Building blocks of existing methods*, Computational Mechanics, 41 (2008), pp. 279–291.

[6] W. Hackbush, *On the computation of approximate eigenvalues and eigenfunctions of elliptic operators by means of a multigrid method*, SIAM Journal on Numerical Analysis, 16 (1979), pp. 201–215.

[7] G. Karypis and V. Kumar, *Metis - unstructured graph partitioning and sparse matrix ordering system, version 2.0*, tech. rep., Department of Computer Science, University of Minnesota, 1995.

[8] Z. Li, Y. Saad, and M. Sosonkina, *pARMS: a parallel version of the algebraic recursive multilevel solver*, Numerical Linear Algebra with Applications, 10 (2003), pp. 485–509.

[9] P. Lindsay and M. Parks, *A Multi-timestepping Extenstion to the Peridynamics Theory*, in 2013 CSRI Summer Proceedings, Sandia National Labs, 2013.

[10] C. Liu, M. H. Jamal, M. Kulkarni, A. Prakash, and V. Pai, *Exploiting domain knowledge to optimize parallel computational mechanics codes*, in Proceedings of the 27th International ACM Conference on International Conference on Supercomputing, ICS '13, New York, NY, USA, 2013, ACM, pp. 25–36.

[11] J. Michopoulos, C. Farhat, and J. Fish, *Modeling and simulation of multiphysics systems*, Journal of Computing and Information Science in Engineering, 5 (2005), p. 198.

[12] N. M. Newmark, *A method of computation for structural dynamics*, Journal of Engineering Mechanics, ASCE, 85 (1959), pp. 67–94.

[13] C. Papalukopoulos and S. Natsiavas, *Dynamics of large scale mechanical models using multilevel substructuring*, Journal of Computational and Nonlinear Dynamics, ASME, 2 (2007), pp. 40–51. Transactions of the ASME.

[14] A. Prakash, *Multi-time-step domain decomposition and coupling methods for non-linear structural dynamics*, tech. rep., 2007.

[15] A. Prakash and K. D. Hjelmstad, *A FETI based multi-time-step coupling method for Newmark schemes in structural dynamics*, International Journal for Numerical Methods in Engineering, 61 (2004), pp. 2183–2204.

[16] ———, *A multi-time-step coupling method for Newmark schemes in structural dynamics*, in proceedings of McMat2005: Joint ASME/ASCE/SES Conference on Mechanics and Materials, June 2005.

[17] A. Prakash, E. Taciroglu, and K. D. Hjelmstad, *Computationally efficient multi-time-step method for partitioned time integration of highly nonlinear structural dynamics*, Computers & Structures, 133 (2014), pp. 51–63.

[18] Y. Saad and J. Zhang, *BILUTM: A domain-based multilevel block ILUT preconditioner for general sparse matrices*, SIAM Journal on Matrix Analysis and Applications, 21 (1999), pp. 279–299.

[19] S. Silling and E. Askari, *A meshfree method based on the peridynamic model of solid mechanics*, Computers & Structures, 83 (2005), pp. 1526 – 1535.

IMPROVING THE TRACER CORRELATION PROBLEM IN A SPECTRAL ELEMENT DYNAMICAL CORE

NICOLAS A. LOPEZ* AND MARK A. TAYLOR†

Abstract. The existence of a lack of linear correlation (LC) preservation is confirmed and improved in the tracer transport scheme of the spectral element core used by the Community Atmosphere Model (CAM). Application of a certain simple tracer chemistry reaction before each model time step can test whether the scheme actually preserves linear tracer correlations to machine precision. Using this method, we confirm previous results that, at times, the shape-preserving filters used in the transport scheme do not preserve LCs. However, it was found that the initial conditions do not preserve LCs to quite machine precision, and there is other evidence that the limiter is facilitating the accumulation of this roundoff error. In any case, limiter subroutine is modified so that the correlation does not break significantly for a 12 day simulation. This process is done outside of CAM, and instead, directly within a shallow water model that uses the HOMME dynamical core.

1. Introduction. In addition to mass conservation, tracer transport schemes often attempt to preserve physical bounds or monotonicity. However, the transport scheme is also meant to satisfy a few other properties regarding pairs of tracers [4]. This includes the ability to maintain linear tracer correlations (LCs) to within machine precision for the length of any simulation, when given a pair of related tracers. The method known as "toy chemistry" testing put forth by [3] attempts to look at how well the spectral element core of the Community Atmosphere Model (CAM-SE) handles chemistry interactions at the sub-grid level. By implementing a simple chemistry reaction at every time step of the model, the toy chemistry tests the scheme's ability to preserve mixing ratio correlation between the two tracers at every node and time-step. If a scheme fails to do this, the scheme's implementation should be re-examined.

To ensure realistic problem solutions, monotinicity can be enforced in CAM-SE by limiting the amount of fluctuation between the nodes of adjacent elements. The optimization-based limiter described in [1] (here known as the Equal Value or EV limiter) attempts to reduce unphysical fluctuations while minimizing the discrete L_2 error norm. In addition to conserving mass, this particular limiter was thought to have the advantage of preserving LCs. However, [3] used the toy chemistry test to show that the limiter does not always preserve LCs in its present implementation. This prompted the re-investigation into the limiter code and launched the attempt to fix the issue.

2. HOMME Model and Transport Scheme. The High Order Method Modeling Environment (HOMME) is the spectral element dynamical core of the Community Atmosphere Model (CAM) that uses the continuous Galerkin spectral element method [6] for unstructured quadrilateral meshes. For advection, it looks similar to the finite-volume method in that it conserves mass during transport and employs the same Lagrangian discretization in the vertical. Second-order accurate Runge-Kutta methods are used for the time-stepping. The HOMME shallow water model originally solves the continuity equation written as

$$\frac{\partial(\rho\phi)}{\partial t} = -\nabla \cdot (\mathbf{v}\rho\phi) \tag{2.1}$$

without chemistry or any external forcing. The continuity equation preserves mixing ratio (ϕ) along the flow if density is calculated at each step [2]. For our experiments, it is important

*Florida State University, nal10c@my.fsu.edu
†Sandia National Laboratories, mataylo@sandia.gov

that ϕ be preserved by the advection scheme, so that the linearly correlated chemistry is the only localized effect along the path of the tracer.

At each time step, the toy chemistry forcing is applied directly to the weak formulation nodes of each element before the full advection step. In HOMME, there is the option to apply a limiter to the nodes after hyperviscosity and the local element time-step have been applied. In our experiments, the application of this limiter (if it is indeed applied) is the last significant change to the elements before output functions are called and the next time step takes place. For our simulations, no physics (other than chemistry) are applied and constant hyperviscocity is used. As for the simulations in our research, each run is done using Gauss-Lobatto weights over 16 nodes on 20 elements. We chose a time step of 432 seconds to be run over a period of 12 days with a viscocity coefficient of 2.74×10^{14}. These values allowed for realisitc chemistry forcing terms. Finally, the output are interpolated to a 192×384 lat/lon grid.

3. Equal Value Limiter. The act of smoothing numerically-introduced oscillations (such as those caused by CFL condition breaches) is often done with what is known as a quasi-monotone limiter. This is done to provide a more realistic physical solution at the cost of introducing a bit of error into the system. There are various options of monotinicity-preserving limiters within HOMME, but [3] showed how the default limiter in CAM-SE can fail the terminator test. This limiter takes in Gaussian weights as well as the weak formulation values at the nodes (q) multiplied by density (ρq) and shrinks the range of q to be within a given minimum/maximum. We can symbolize the limiting process in terms of a function lim, where $\tilde{q} = lim(q, q_{max}, q_{min})$. These q_{min} and q_{max} quantities can represent a physical limit (non-zero) or a function of neighboring element's min/max. However, simply truncating the values to be within the range will violate mass conservation, unless the mass change is re-distributed among the other nodes. This can be written as

$$\int \rho\tilde{q} = \int \rho q \tag{3.1}$$

This redistribution can be done many different ways. Minimizing the L_2 error norm seems an obvious criteria for the redistribution scheme, but it is also important to remember to maintain correlation with a linearly related tracer. When combined with 3.1, we describe the L_2 optimization as

$$\min \int_{elem} \rho(\tilde{q} - q)^2 \tag{3.2}$$

or in discrete form using Gaussian weights w_i ...

$$\begin{aligned} &\min_{\tilde{q}\in R} \sum_{i=1}^{np} w_i\rho_i(\tilde{q}_i - q_i)^2 \\ &\text{with the constraints} \\ q_{min} \le \tilde{q}_i \le q_{max} \quad \text{and} \quad &\sum_{i=1}^{np} w_i\rho_i\tilde{q}_i = \sum_{i=1}^{np} w_i\rho_i q_i. \end{aligned} \tag{3.3}$$

Here, np denotes the total number of nodes in the element, and q_i is a node value on a single tracer.

In the EV limiter, all out-of-bounds values are brought up/down to q_{min} or q_{max}. This new set of truncated values is notated as q_i'. The amount of mass change that this generates is assigned as

$$\alpha = \sum w_i(q_i - q_i') \tag{3.4}$$

This is the amount of mass that must be redistributed across the nodes and can be negative just as easily as it can be positive. Solving the minimization problem (ignoring the constraints) reveals that the update to node q_i' can be written as

$$\tilde{q}_i = q_i' + \frac{\alpha}{\sum_i w_i} \quad \text{for every } i \text{ s.t.} \begin{cases} q_i' \neq q_{max} \text{ if } \alpha > 0 \\ q_i' \neq q_{min} \text{ if } \alpha < 0 \end{cases} \tag{3.5}$$

This process is repeated on intermediate result $\tilde{q}_i$ until the maximum number of iterations occurs or the constraints are met ($\alpha = 0$). It can be shown that no matter how many iterations are performed, the amount of mass in the element does not change. More important to our experiment, if a second set of tracer mixing ratio nodes q_y is defined as a linear operation on original tracer q_x by the equation

$$q_y = Aq_x + B, \tag{3.6}$$

then doing this same linear operation on $\tilde{q}_x$ should produce $\tilde{q}_y$. This is defined as preserving LCs with respect to mixing ratios. To prove that the EV limiter theoretically preserves this correlation, we must first assume that the truncation step does too (i.e. $q_y' = Aq_x' + B$). This is a reasonable in HOMME, because q_{min} and q_{max} come from interdependent neighboring element nodes. From there, we can show the linear dependence of the two α masses:

$$\begin{aligned} \alpha_x &= \sum w(q_x - q_x') \\ \alpha_y &= \sum w(q_y - q_y') \\ &= \sum w((Aq_x + B) - (Aq_x' + B)) \\ &= A \sum w(q_x - q_x') \\ &= A(\alpha_x) \end{aligned} \tag{3.7}$$

We then use Equation 3.5 to define EV updated tracer nodes:

$$\begin{aligned} \tilde{q}_x &= q_x' + \frac{\alpha_x}{\sum w} \\ \tilde{q}_y &= q_y' + \frac{A(\alpha_x)}{\sum w} \end{aligned} \tag{3.8}$$

Simply solving for q_x' and q_y' and using Equation 3.6 cancels the term with α:

$$\begin{aligned} \tilde{q}_y - A(\frac{\alpha_x}{\sum w}) &= A(\tilde{q}_x) - A(\frac{\alpha_x}{\sum w}) + B \\ \tilde{q}_y &= A(\tilde{q}_x) + B \end{aligned} \tag{3.9}$$

We have shown that the EV limiter theoretically preserves LCs, but the results of the terminator test in HOMME reveal that its implementation can fail at the terminator, where there exists a sharp gradient of tracer values.

4. Toy Chemistry Implementation. In terms of real-world application, the toy chemistry simulates the photolysis of Cl2 in the stratosphere when the sun begins to rise on the horizon. This reaction experienced near the Earth's solar terminator is simulated by

$$\begin{aligned} \mathrm{Cl_2} &\xrightarrow{k_1} 2\mathrm{Cl} \\ \mathrm{Cl} + \mathrm{Cl} &\xrightarrow{k_2} \mathrm{Cl_2} \end{aligned} \tag{4.1}$$

Converting these reactions to kinetic equations and rearranging, we obtain

$$\begin{aligned} \frac{dCl}{dt} &= 2\mathrm{k_1Cl_2} - 2\mathrm{k_2ClCl} \\ \frac{dCl_2}{dt} &= -\mathrm{k_1Cl_2} + \mathrm{k_2ClCl} \end{aligned} \tag{4.2}$$

In these equations, the reaction rates k_1 and k_2 are global fields that do not change with time. The equation for the k_1 rate of Chlorine molecule decomposition is set to a physically realistic field centered around (λ_c, θ_c) and shown in 4.1 according to the following equation:

$$k_1(\lambda, \theta) = \max[0, \sin\theta \sin\theta_c + \cos\theta \cos\theta_c \cos(\lambda - \lambda_c)] \ , \tag{4.3}$$

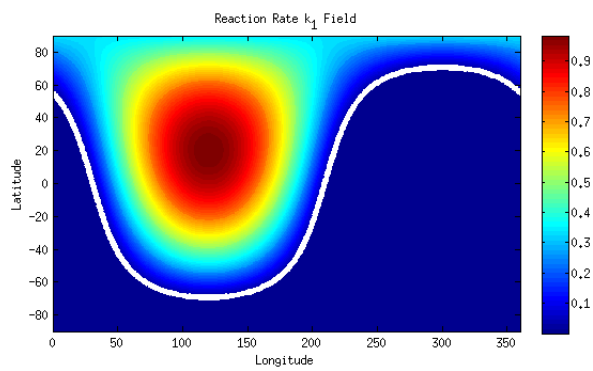


Fig. 4.1: The reaction rate k_1 that stays constant throughout all simulations. The thick white line marks the solar terminator, where k_1 drops to zero. The other reaction rate $k_2 = 1$ for all locations and times.

Additionally, the natural rate of chlorine molecule formation is simulated with a constant $k_2 = 1$ across the whole globe. If we were to initialize the mixing ratios to be $Cl_2 = 0$ and $Cl = 4.0 \times 10^{-6}$ and then apply the chemistry shown in Equation 4.2, then it can be shown that the steady-state solution fields would become

$$\begin{aligned} \lim_{t\to\infty} \mathrm{Cl(t)} &= \mathrm{D - r} \\ \lim_{t\to\infty} \mathrm{Cl_2(t)} &= \frac{1}{2}(\mathrm{Cl_z - D + r}) \\ &\text{where} \\ \mathrm{Cl_z = Cl + 2Cl_2}, \qquad \mathrm{r} &= \frac{\mathrm{k_1}}{4\mathrm{k_2}}, \qquad \mathrm{D} = \sqrt{\mathrm{r^2 + 2rCl_z}} \end{aligned} \tag{4.4}$$

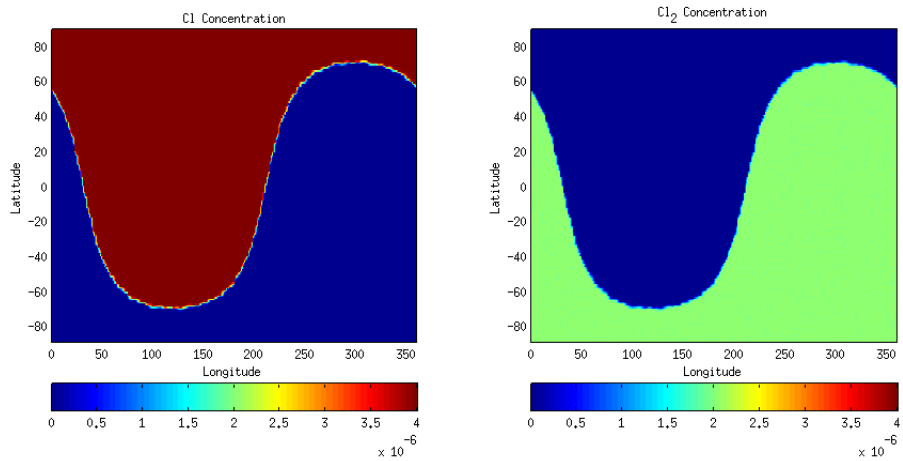


Fig. 4.2: Initial conditions for Cl and Cl_2 tracers

These fields shown in Figure 4.2 will actually be our initial conditions for the toy-chemistry simulation. The boundary between the gases initially lies at the solar terminator. Although the tracer gases are advected across the terminator in the simulations, the terminator is an artifact of the reaction constants, which do not change with time. If we superimpose the plots in Figure 4.2, we can see that $Cl + 2Cl_2 = 4e^{-6}$ over the whole domain. Furthermore, our chemistry equations 4.1 indicate that the total number of Chlorine atoms should be conserved throughout the duration of any simulation. That is ...

$$\frac{DCl_z}{Dt} = \frac{D}{Dt}[\mathrm{Cl} + 2\mathrm{Cl}_2] = 0 \ . \tag{4.5}$$

Given this, we can expect $Cl_z = 4e^{-6}$ to be preserved to machine precision over the course of the simulation. The two different tracers Cl and Cl_2 are assigned to 2 independent levels of the HOMME shallow water model. At each time step, the tracer concentration values associated with each element interact with one another, but only by means of chemsistry. This interaction can be done by first calculating analytical forcing terms

$$\begin{aligned} F_{Cl} = -L_{\Delta t}\frac{(Cl - D + r)(Cl + D + r)}{1 + E(\Delta t) + \Delta t L_{\Delta t}(Cl + r)} \\ F_{Cl_2} = -\frac{1}{2}F_{Cl} \\ \text{where} \\ L_{\Delta t} = \begin{cases} \frac{1-e^{-4k_2 D\Delta t}}{D\Delta t} & \text{if } D > 0 \\ 4k_2 & \text{if } D = 0 \end{cases} \end{aligned} \tag{4.6}$$

These forcing terms are meant to be applied to ρq, instead of mixing ratio nodes q, because this the variable being used by the advection scheme. The calculated density ρ is the same for both tracers. To prove that the chemistry itself preserves LCs, we first define

tracer updates $\hat{q}_{Cl}$ and $\hat{q}_{Cl2}$:

$$\begin{aligned} \rho\hat{q}_{Cl} &= \rho q_{Cl} + F_{Cl} \\ \rho\hat{q}_{Cl2} &= \rho q_{Cl2} - \frac{1}{2}F_{Cl} \end{aligned} \tag{4.7}$$

Solving for q_{Cl} and q_{Cl2} and then plugging into Equation 3.6 using $A = -\frac{1}{2}$ and $B = 2e^{-6}$, we get

$$\begin{aligned} \hat{q}_{Cl2} + \frac{F}{2\rho} &= -\frac{1}{2}\hat{q}_{Cl} + \frac{F}{2\rho} + 2e^{-6} \\ \hat{q}_{Cl2} &= A\hat{q}_{Cl} + B \end{aligned} \tag{4.8}$$

This shows that the chemistry update preserves a correlation. A little more algebra shows that the initial tracer condition stated in Equation 4.4 does as well. Since the chemistry preserves LCs, it is up to the scheme and the limiter to also preserve the LC. The forcing terms can then be applied to the ρq values at each element node using the following coding assignment:

```
rhoPhi_cl = rhoPhi_cl + FCl*dt
rhoPhi_cl2 = rhoPhi_cl2 + FCl2*dt
```

Once this chemistry has been applied to the two tracer levels in the model, the transport scheme advancement is called, using a prescribed non-divergent flow.

5. Running Toy Chemistry in HOMME. There is a wide variety of prescribed flows that could be used in the shallow water model to test how well the model preserves linear tracer correlations, but for the purposes of replicating [3]'s results, the reversing deformational flow used by [5] is used. This flow consists of two global gyres that shift eastward with time and re-appear in the west with periodic boundary conditions. With this flow, tracers will actually return to their initial positions at prescribed times after a significant amount of perturbation. Of course this advantage of having the analytical solution at the final time step only applies to a chemistry-free model, so we cannot calculate error norms for the Cl and Cl_2 tracer fields. Instead we know Equation 4.5, so comparing our Cl_z field to a constant 4×10^{-6} field will be our error criteria. More specifically, [3] defines $L_2(t)$ and $L_\infty(t)$ error norms to be:

$$\begin{aligned} L_2(t) &= \sqrt{\frac{I[(Cl_z(t) - Cl_z(0))^2]}{I[(Cl_z(0))^2]}} \\ L_\infty(t) &= \frac{max_{\forall\lambda,\theta} \mid Cl_z(t) - Cl_z(0) \mid}{max_{\forall\lambda,\theta} \mid Cl_z(0) \mid} \\ &\text{where} \\ I(\phi) &= \frac{1}{4\pi}\int_0^{2\pi}\int_{-\pi/2}^{\pi/2} \phi(\lambda,\theta,t)cos\theta d\lambda d\theta \ . \end{aligned} \tag{5.1}$$

The ability to determine if the scheme fully preserves LCs can also be done qualitatively by plotting the global Cl_z field. In fact this method is more useful, because the *locations* of the lack of preservation can also be determined. This is important, because the solar terminator is where most of the chemistry occurs.

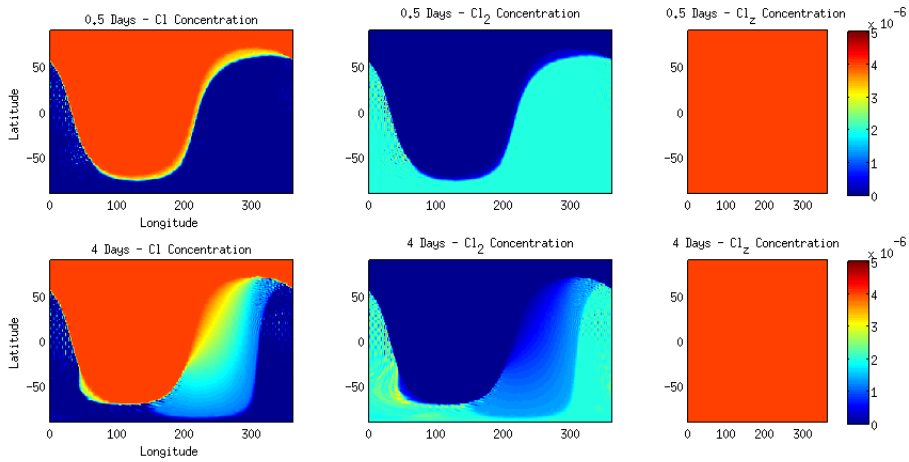


Fig. 5.1: Results of unlimited simulation at 0.5 and 4 days. A uniform Cl_z field indicates a truly semilinear transport operator was used.

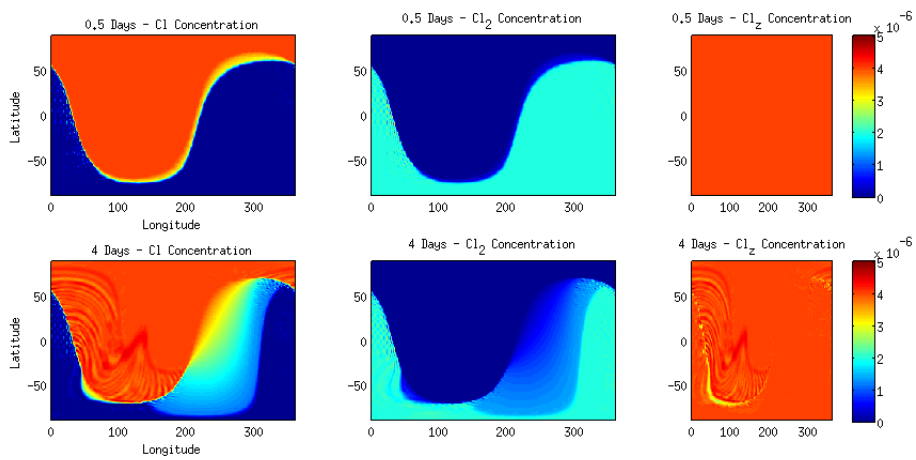


Fig. 5.2: Same figure as 5.1, but with the EV limiter applied. At 0.5 days the underlying Cl_z error is very low, but by 4 days it has grown enough to be significant.

Again [3] claimed that the unlimited advection scheme preserved LCs to machine precision throughout the entire domain, and when the Equal Value limiter was applied, the correlation broke by up to 25% near the terminator. However, we noticed that the no-limiter LC was in fact not preserved to machine precision over the whole domain, even in the initial condition. The Cl_2-dominant region had a machine-precision relative error of Cl_z to $O(10^{-16})$, but the Cl-dominant region preserved only to $O(10^{-12})$. This difference aside, we came extremely close to replicating the no-limiter experiment, but had a feature far removed from the terminator that around Day 10 began to disrupt the Cl_z L_2 norm to above $O(10^{-10})$. Our EV limiter experiment did indeed break the Cl_z preservation much

more drastically, however. Much like [3]'s results, this Cl_z error occured mainly at the terminator and then propogated around the domain as if it were a separate tracer. Figure 5.2 shows that the Equal Value limiter does fairly well through 12 hours, but by Day 4 large errors take up a significant part of the domain. This is in comparison to Figure 5.1, where no limiter is applied.

The authors of [3] suggest that the error at the terminator is generated when the limiter must resolve a very sharp tracer gradient. To verify this, we take the numerical gradient $\sqrt{\frac{\partial Cl}{\partial x}^2 + \frac{\partial Cl}{\partial y}^2}$ at t=12 hours at each gridpoint using a centered difference. Figure 5.3 shows that the Cl_z error does in fact occur at the locations of highest gradient, but the limiter successfully makes the tracer transition smoother and more monotonic, despite its error-producing properties in subsequent time steps.

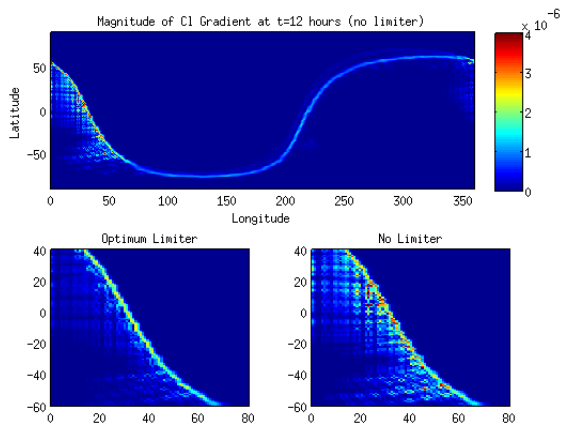


Fig. 5.3: Numerical gradient of Cl a few time steps before the Cl_z relative error begins to reach $O(10^{-2})$ using the EV limiter. The smaller subplots are simply zoomed into the error-prone region of the terminator.

Interestingly, if we increase the initial Cl_z value of $4e^{-6}$ by a few orders of magnitude, we do not see the EV limiter significantly break the correlation. In fact, Figure 5.4 shows that making this change generates a large jump in LC preservation. Another interesting result shown in this figure is the "layering" of the error with respect to initial Cl_z values. The reader is reminded that these are *relative* errors and by this property, should lie on top of one another if they preserve close to machine precision. The fact that they do not is direct evidence of the roundoff error inherent in the calculations used to generate the initial conditions. Another indication of roundoff error is from the lack of exact preservation of the Cl-dominant region of the initial condition mentioned previously. We will leave this roundoff evidence aside for the time being, and instead discuss adjusting the limiter.

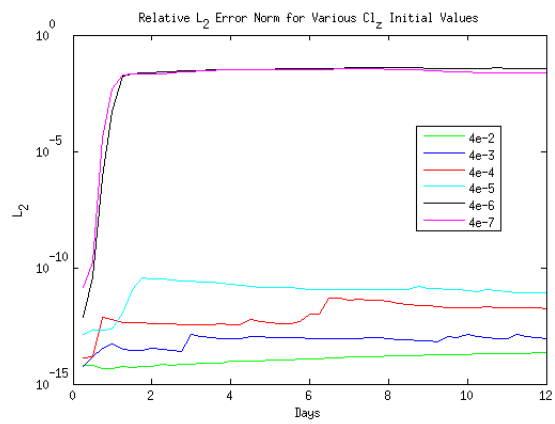


Fig. 5.4: Results of running 12-day simulations with the EV limiter, using various other initial Cl_z values.

6. Resolving the Correlation Problem. Before beginning to resolve the problem, the code was adjusted and streamlined, but these changes did not improve the results. The limiter implementation is relatively simple, but there are certain sub-parts that can be LC preserving, even when the remainder of the limiter is not. For example, we show in Figure 6.1 that when we take the first step of the limiter and only truncate the values within each element without redistributing the mass, the LC between tracers is preserved. It is important to distinguish conserving local mass from preserving linear correlations. The fact that this truncation step preserves LCs to machine precision is a very big result for our problem. It guarantees that the lack of preservation problem *must* lie in the redistribution of mass to the element nodes.

The truncation step alone is obviously not satisfactory for local mass conservation, however, so the need to redistribute the mass that was removed/added still exists. We put forth a few alternative limiters that are also theoretically both mass and LC preserving. One of them applies a consistent change to *all* nodes in the element (All Equal Value or AEV Method), while the other applies consistent *mass* to the same nodes that the EV limiter uses (Equal Mass or EM Method). The update to the truncated values in the EM is as follows:

$$\tilde{q_i} = q_i' + \frac{\alpha}{n(w_i)}$$
$$\text{where } n \text{ is the total number of } i\text{'s s.t. } \begin{cases} q_i' \neq q_{max} \text{ if } \alpha > 0 \\ q_i' \neq q_{min} \text{ if } \alpha < 0 \end{cases} \tag{6.1}$$

The proofs that this method along with the AEV method preserve LCs are similar to Equations 3.8 and 3.9, in that the proofs also have cancellation of α terms. Table 6.1 provides this list of alternatives to the redistribution scheme proposed by the EV limiter.

The AEV method can require many iterations, but the maximum of iterations in our simulations is set to 15, because this number of iterations was found to always bring the amount of mass above/below the max/min to below machine precision. One red flag with these multi-iteration methods is the possiblity of an inconsistent number of iterations across both tracers. For example, if the limiter were to perform 2 iterations on Cl and 3 iterations

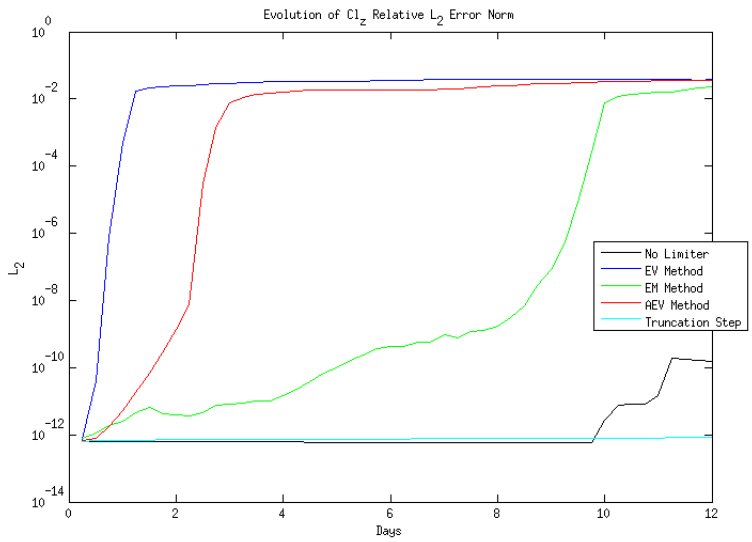


Fig. 6.1: The L_2 norms associated with Cl_z from Equation 5.1 for different mass redistribution schemes. The truncation method did not redistribute any truncated mass.

on Cl_2 due to stopping criteria that is not LC preserving, it can be seen how LC would not be theoretically preserved for that element. However, we found that even when only one iteration is performed, the LC preservation problem still exists. Figure 6.1 shows that EM limiter provides significant improvement over the EV limiter out to approximately Day 10. Eliminating the LC preservation error was a main goal of this research, and in a sense, it was acheived. This limiter should produce a bit more discrete L_2 error in ϕ than the EV limiter, but it was a good alternative for our toy chemistry problem.

Method	Description	Typical Number of Iterations	Conserves Local Mass
Equal Value (EV) Method	Spread positive α to all nodes that are not at maximum or negative α to all nodes that are not at minimum. Each node is changed by the same amount. (See Equation 3.5)	$O(10^0)$	Yes
All Equal Value (AEV) Method	Much like the EV method, the same amount of change is applied, but to *all* nodes ... even those already at the prescribed max/min	$O(10^1)$	Yes
Equal Mass (EM) Method	Distributes mass to the same nodes as the EV method, but each node receives the same amount of *mass*, instead of the same change in value. (See Equation 6.1)	$O(10^0)$	Yes
Truncation	Simply reset the node values that are outside the bounds to the max/min. No further adjustment is done to the values.	1	No

Table 6.1: Various ways of redistributing truncated mass inside the limiter. All methods generate L_2 tracer error (not necessarily Cl_z error), but the EV limiter theoretically generates the least.

7. Conclusions and Future Work. We have provided an alternative to the Equal Value limiter that does reasonably well out to 12 days, but we know that it can do much better. We know this because in our test case, applying only the limiter truncation step (or no limiter at all) conserves our sum of linearly correlated tracers (Cl_z) to nearly machine precision at the terminator. However, we postulate that the unexpected behavior displayed at Day 10 for the simulation without the application of the limiter may have been the result of the previously described roundoff error artifacts. This would explain why [3] obtained different results. Future research needs to determine if the limiters are simply exacerbating the small breaks in correlation associated with roundoff error in the initial condition. The problem may be unresolvable, and may just be a by-product of the toy chemistry methodology itself. We also intend to discover the discrete error norm that the Equal Mass limiter optimizes, what it can be used for, and why this limiter does so much better with the terminator toy chemistry case.

REFERENCES

[1] O. Guba, M. Taylor, and A. St-Cyr, *Optimization-based limiters for the spectral element method*, Journal of Computational Physics, 267 (2014), pp. 176 – 195.

[2] P. Lauritzen, P. Ullrich, and R. Nair, *Atmospheric transport schemes: Desirable properties and a semi-lagrangian view on finite-volume discretizations*, in Numerical Techniques for Global Atmospheric Models, P. Lauritzen, C. Jablonowski, M. Taylor, and R. Nair, eds., vol. 80 of Lecture Notes in Computational Science and Engineering, Springer Berlin Heidelberg, 2011, pp. 185–250.

[3] P. H. Lauritzen, A. J. Conley, J.-F. Lamarque, F. Vitt, and M. A. Taylor, *The terminator "toy" chemistry test: a simple tool to assess errors in transport schemes*, Geoscientific Model Development, 8 (2015), pp. 1299–1313.

[4] P. H. Lauritzen and J. Thuburn, *Evaluating advection/transport schemes using interrelated tracers, scatter plots and numerical mixing diagnostics*, Quarterly Journal of the Royal Meteorological Society, 138 (2012), pp. 906–918.

[5] R. D. Nair and P. H. Lauritzen, *A class of deformational flow test cases for linear transport problems on the sphere*, Journal of Computational Physics, 229 (2010), pp. 8868 – 8887.

[6] M. A. Taylor and A. Fournier, *A compatible and conservative spectral element method on unstructured grids*, Journal of Computational Physics, 229 (2010), pp. 5879 – 5895.

FIRST-ORDER APPROXIMATE AUGMENTED LAGRANGIAN METHOD (FOAAL) IMPLEMENTED VIA AN OBJECT-PARALLEL INFRASTRUCTURE FOR FIRST-ORDER METHODS

GYÖRGY MÁTYÁSFALVI*, JONATHAN ECKSTEIN†, AND JEAN-PAUL WATSON‡

Abstract. We describe the early stages of the implementation of a general purpose nonlinear solver. This effort is part of an open-source software project tentatively named FOAAL (First-Order Approximate Augmented Lagrangians). Our solver is based on a classical augmented Lagrangian method with a novel inexact solution condition for the subproblems. The source-code is written in C++ using a special vector-manipulation substrate [6]. At present, FOAAL only works in serial mode. However, our implementation permits a ready parallelization of the optimization algorithm, since switching between serial and parallel mode does not require any changes in the algorithm's source code. We also provide infrastructure to support line-search methods, and interfaces to various modeling languages, such as AMPL [8] or PYOMO [14], in a transparent manner that does not "clutter" the principal implementation. In the future we would like to demonstrate the usefulness of our vector-substrate tools by employing a serial version of FOAAL to solve a real-world Unit Commitment problem, after that we plan to develop a parallel Unit Commitment "solver" using FOAAL.

Acknowledgments. This research was supported by Sandia National Laboratories and by National Science Foundation grant CCF-1115638.

1. Introduction. The overall motivation behind the object-parallel implementation of FOAAL is to exploit ever-increasing computer power to solve larger continuous optimization problems, or to solve problems faster so that their solution may be embedded in other algorithms (for example, to handle discrete variables). The challenge in this field is that while computing technology continues to grow in power, most gains are now coming through increased parallelism and proliferation of processor cores, rather than acceleration of individual instruction streams. Augmented Lagrangian algorithms, especially when using recent theoretical advances, should provide a relatively simple and elegant way to adapt to these trends in computer architecture. Nonetheless efficient implementation of numerical algorithms in parallel computing environments can be challenging. The same underlying algorithm may have to be re-implemented multiple times to adapt to different hardware environments or applications, and the resulting code may be difficult to read. A natural route to more elegant and portable implementation of parallel algorithms is to use established object-oriented programming concepts. We hope that this approach will differentiate itself from other existing frameworks such as TAO (Toolkit for Advanced Optimization), which is a C based method developed at Argonne National Laboratories, and other open-source or proprietary optimization software that are less readily customizable by the user.

This paper leads through the steps and techniques taken to implement the serial version of FOAAL using our object-parallel framework. The algorithms proposed here leverage proven, successful methods. In addition we will use a new relative error criteria [7] to determine with how much precision to solve the nonlinear subproblems produced by the augmented Lagrangian algorithm. Our approach will be to use first-order methods [11, 12, 13, 2, 3] that are Hessian-free, and designed to be able to take advantage of parallel computer architectures in a more straightforward way than competing methods. However, theoretically FOAAL is not constrained to first-order methods to solve the subproblems. One may consider other algorithms for the subproblems if the objective function and constraint properties allow it. This is also supported by the design of FOAAL, which allows for

*Rutgers University - Doctoral Program in Operations Research, matyasfalvi@gmail.com
†Rutgers University - Department of MSIS and RUTCOR, jeckstei@rci.rutgers.edu
‡Sandia National Laboratories - Discrete Math & Optimization, jwatson@sandia.gov

a ready substitution of the subproblem solver. The result would be a general-purpose nonlinear optimization solver framework for parallel environments, capable of handling general nonlinear constraints.

As a test case, the methodology will be applied to the extensive form of stochastic programming problems, in particular to the unit commitment problem. Since FOAAL is an open-source project we hope that the unit commitment example will encourage both academic and industry professionals to use our tools to develop parallel solvers for other applications of special interest.

The remainder of this report is organized as follows: Section 2 Introduces the theoretical background of FOAAL and the relative error criterion. Section 3 describes the actual implementation of FOAAL and how elements of our infrastructure that are specific to optimization problems are used in that process. Finally, in Section 4 we introduce the unit commitment problem and go through our plan on how to extend our tools to solve the unit commitment problem in parallel.

2. FOAAL algorithm. We consider a general continuous optimization problem of the form

$$\begin{array}{ll} \min\limits_{x\in\mathbb{R}^n} & f(x) \\ \text{ST} & g_i(x) \in K_i, i \in 1..m \end{array} \tag{2.1}$$

Here, $f : \mathbb{R}^n \to \mathbb{R}$ and $g_i : \mathbb{R}^n \to \mathbb{R}^{d_i}$ $(i \in 1..m)$ are continuously differentiable functions, and $K_i \subset \mathbb{R}^{d_i}$ $(i \in 1..m)$ are closed convex cones. For example, $d_i = 1$ and $K_i = \{0\}$ corresponds to a standard equality constraint $g_i(x) = 0$, while $d_i = 1$, $K_i = (-\infty, 0]$ corresponds to a conventional inequality constraint $g_i(x) \leq 0$. Other common constraint forms, such as semidefinite matrix constraints, can be readily modeled in this format. Now consider the following convex optimization problem, which is a special case of the above:

$$\begin{array}{ll} \min & f(x) \\ \text{ST} & h(x) = 0 \\ & g(x) \leq 0. \end{array} \tag{2.2}$$

where $f : \mathbb{R}^n \to \mathbb{R}$ is convex, $h : \mathbb{R}^n \to \mathbb{R}^{m_1}$ is affine, and $g(x) = (g_1(x), \ldots, g_{m_2}(x))$, where $g_1, \ldots, g_{m_2} : \mathbb{R}^n \to \mathbb{R}$ are convex, and f and g are differentiable. We will continue the analysis of FOAAL by using the problem formulation in (2.2). The augmented Lagrangian dual function $L_D(\lambda, \mu)$ of (2.2) takes the following form:

$$\inf_{x\in\mathbb{R}^n} \left\{ f(x) + \langle \lambda^{k-1}, h(x)\rangle + \frac{c_k}{2} \|h(x)\|^2 + \frac{1}{2c_k} \left\|\max\left\{0, \mu^{k-1} + c_k g(x)\right\}\right\|^2 \right\} \tag{2.3}$$

Applying the *proximal point algorithm* (PPA) to an operator derived from the dual of (2.2), we obtain as described in [18] the *method of multipliers*:

$$x^k \in \operatorname*{Arg\,min}_{x\in\mathbb{R}^n} \left\{ f(x) + \langle \lambda^{k-1}, h(x)\rangle + \frac{c_k}{2} \|h(x)\|^2 + \frac{1}{2c_k} \left\|\max\left\{0, \mu^{k-1} + c_k g(x)\right\}\right\|^2 \right\} \tag{2.4}$$

$$\lambda^k = \lambda^{k-1} + c_k h(x^k) \tag{2.5}$$

$$\mu^k = \max\left\{0, \mu^{k-1} + c_k g(x^k)\right\} \tag{2.6}$$

where c_k is a sequence of scalars with $\inf_k \{c_k\} \geq 0$, the "max" operations are interpreted componentwise, $\lambda^{k-1} \in \mathbb{R}^{m_1}$ and $\mu^{k-1} \in \mathbb{R}^{m_2}_+$ are the previous iteration's estimates of the

Lagrange multipliers for the equality and inequality constraints in (2.2). And $\lambda^k \to \hat{\lambda}$, $\mu^k \to \hat{\mu}$ converge to a root of the subgradient of $-L_D(\lambda, \mu)$ i.e. $\nabla L_D(\hat{\lambda}, \hat{\mu}) = 0$.

Under constraint qualification $\nabla L_D(\hat{\lambda}, \hat{\mu}) = 0$, $\hat{\lambda} \geq 0$, $g(x^*) \leq 0$, $h(x^*) = 0$, and $\langle \lambda, g(x^*) \rangle = 0$, where

$$x^* \in \operatorname*{Arg\,min}_{x \in \mathbb{R}^n} \left\{ f(x) + \langle \hat{\lambda}, h(x) \rangle + \frac{c_*}{2} \|h(x)\|^2 + \frac{1}{2c_*} \|\max\{0, \hat{\mu} + c_* g(x)\}\|^2 \right\} \tag{2.7}$$

yield the necessary conditions for local optimality of x^* to (2.2) according to the *Karush-Kuhn-Tucker* (KKT) conditions. For a detailed development of the FOAAL algorithm including convergence proofs refer to [7].

In summary the general approximation framework described in [7] produces the following set of recursive conditions, with arbitrary starting values $\lambda^0 \in \mathbb{R}^{m_1}$, $\mu^0 \in \mathbb{R}^{m_2}_+$ and $w^0 \in \mathbb{R}^n$:

$$y^k = \nabla_x \left[f(x) + \langle \lambda^{k-1}, h(x) \rangle + \tfrac{c_k}{2} \|h(x)\|^2 + \tfrac{1}{2c_k} \left\| \max\left\{0, \mu^{k-1} + c_k g(x)\right\} \right\|^2 \right] \tag{2.8}$$

$$\tfrac{2}{c_k} \left| \langle w^{k-1} - x^k, y^k \rangle \right| + \left\| y^k \right\|^2 \leq \sigma \left(\left\| h(x^k) \right\|^2 + \left\| \min\left\{ \tfrac{1}{c_k} \mu^{k-1}, -g(x^k) \right\} \right\|^2 \right) \tag{2.9}$$

$$\lambda^k = \lambda^{k-1} + c_k h(x^k) \tag{2.10}$$

$$\mu^k = \max\left\{0, \mu^{k-1} + c_k g(x^k)\right\} \tag{2.11}$$

$$w^k = w^{k-1} - c_k y^k \tag{2.12}$$

Here, (2.8) and (2.9) replace the exact augmented Lagrangian minimization (2.4); the "max" and the "min" in (2.8) and (2.9) respectively are interpreted componentwise.

3. Implementation of FOAAL. In this section we will go over the object-parallel implementation of FOAAL. First by showing parts of the C++ source-code and then by going over it step by step. The method introduced in section 2, when implemented via our object-parallel framework, is illustrated in figure 3.2.

What we see in figure 3.1 is the FOAAL class declaration and in figure 3.2 we have the minimization routine of FOAAL. What makes our approach object-parallel are the following members of FOAAL: `VectorObjects` (`x`, `hx`, `gx`, etc.), the `AbstractProblem` pointer `pr` and the `LineSearchBasedMethod` pointer `boxSolver`. FOAAL's constructor takes two arguments, one `AbstractProblem` pointer and one `LineSearchBasedMethod` pointer. The `AbstractProblem` class is used to initialize the `VectorObject` and the penalty parameter `double* c` members of the FOAAL class. The `LineSearchBasedMethod` is used to solve the subproblems generated by the augmented Lagrangian method. The argument of the minimization routine, an `AbstractTermination` class, is an abstract C++ class and is passed to the `LineSearchBasedMethod`'s minimization routine. Through this unified termination interface we can easily modify the termination conditions for our subproblem solvers.

```
class FOAAL {

  AbstractProblem* pr;
  LineSearchBasedMethod* boxSolver;

  VectorObject x;
  VectorObject hx;
  VectorObject gx;
  VectorObject grad;
  VectorObject lambda;
  VectorObject mu;
  VectorObject muProj;
  VectorObject muFeas;
  VectorObject zeroVec;
  VectorObject w;
  VectorObject wStep;

  double* c;
  double sigma;
  double maxC;
  double factorC;
  double minImprov;
  double prevPhiC;

  double objVal;
  double tol;


  int augIter;
  int maxAugIter;
  int resetCount;

  int boxIter;
  bool kktNotSat;

public:
  FOAAL(AbstractProblem* pr_, LineSearchBasedMethod* boxSolver_);

   virtual ~FOAAL();

   VectorObject& minimize(AbstractTermination& termination);
};
```

Fig. 3.1: Source-code of FOAAL class members.

3.1. VectorObjects and AbstractVectors. The `AbstractVector` class creates a general interface that abstracts both the representation of vectors and the methods used to perform simple mathematical manipulations such as addition and scaling. To avoid cluttering algebraic expression code with excessive pointer dereferencing, we defined the class, called `VectorObject`, which essentially encapsulates a pointer to an `AbstractVector`.

```
VectorObject& FOAAL::minimize(AbstractTermination& termination) {

 while (augIter < maxAugIter && kktNotSat) {
     augIter++;

     /* Solve sub-problem */
     x = boxSolver->minimize(termination);

     pr->objGrad(x, grad);
     pr->equConVal(x, hx);
     pr->inequConVal(x, gx);

     muFeas = ( -1.0 / (*c) ) * mu;
     muProj.max(muFeas, gx);
     double phiC = std::max(hx.normInf(), muProj.normInf());

     /* Check KKT */
     if(phiC < tol && boxSolver->getGradNormInf() < tol) {
         std::cout<<"Terminating outer loop with: ";
         PVAR(phiC)
         kktNotSat = false;
     }

     /* Track accumulated "error drift" */
     if(augIter <= 3) {
         w = x;
     } else {
         w = w - (*c) * grad;
         wStep = w - x;
         if(kktNotSat && wStep.norm2() > 1.0e+2 * (*c) * grad.norm2() \
         && resetCount  < 5) {
             w = x;
             resetCount++;
         }
     }

     /* Update multipliers and parameters */
     lambda = lambda + (*c) * hx;
     mu = mu + (*c) * gx;
     mu = muProj.max(zeroVec, mu);

     /* Update penalty parameter */
     if(augIter > 1 && phiC > std::max(tol, minImprov*prevPhiC)) {
         (*c) = std::min(factorC * (*c), maxC);
     }

     prevPhiC = phiC;
 }
 return x;
}
```

Fig. 3.2: Source-code of FOAAL's minimization method.

When implementing the `AbstractVector` class our basic goal was to use the operator overloading capabilities of the C++ language [20] to be able to express simple algorithmic manipulations of vectors in a concise, readable manner, portable without recoding between different applications, data representations, and hardware platforms. More details about the vector classes such as efficient overloading through delayed evaluation etc. are provided in [6]. The `AbstractVector` class makes it possible to use different vector classes for linear-algebra computations without having to change the optimization algorithm. For example one may readily use the Epetra package, part of the Trilinos project [15], to do linear algebra operations. Because this construct easily allows us to parallelize the underlying linear-algebra of optimization algorithms it is a fundamental building block of our object-parallel framework.

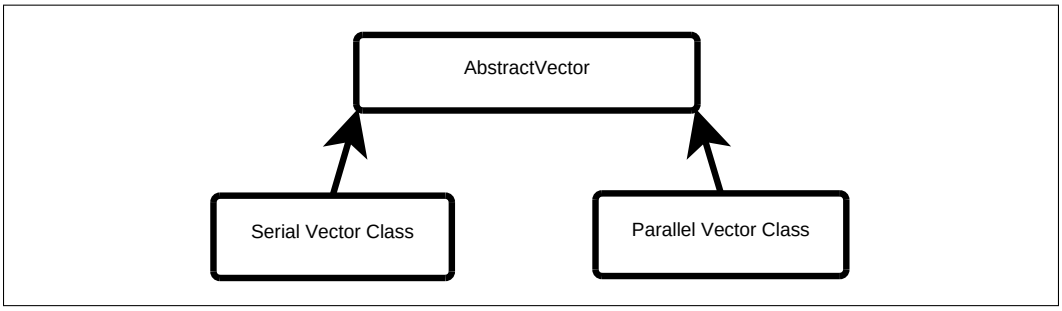


Fig. 3.3: Inheritance of Vector classes

3.2. Optimization-Specific Infrastructure.

3.2.1. The AbstractProblem Class and Its Derived Classes. The `AbstractProblem` class constitutes a unified interface through which our optimization algorithms interact with problem instances and third party modeling languages. One defines a class of problems by deriving a class from `AbstractProblem` (although perhaps indirectly). `AbstractProblem`-derived classes hold problem instance data and contain methods for objective function and gradient evaluation. The problem class also stores upper and lower bounds on variables, when present. `AbstractProblem`-derived classes contain for example methods to represent general constraints, a `VectorObject` that represents the solution algorithm starting point, and abstract methods to generate objects for caching function and gradient values. Through the `VectorObject` representing the algorithm starting point, the `AbstractProblem`-derived class determines the vector class that the solution algorithm uses to execute the linear algebra computations. The algorithm classes initialize their `VectorObject` members by executing "cloning" methods on the initial point provided by the problem class. For the serial version of FOAAL we have created the `AugLagAslProblem` class, which is passed to the constructor of FOAAL. This problem class is derived from the `AslProblem` problem class.

The `AslProblem` accommodates an interface to the AMPL Solver Library [9] (ASL), which in turn provides all the necessary information to create a problem instance in serial. It also contains routines to evaluate the objective function and its gradient as well as the problem constraints and their gradients. The `AslProblem` problem uses ASL routines to read ".nl" [10] files and to write ".sol" files. The ".nl" file format is used by many modeling languages to communicate problem instances to solvers, and the ".sol" file by many solvers to pass the optimal solution back to the modeling environment. Hence the `AslProblem`

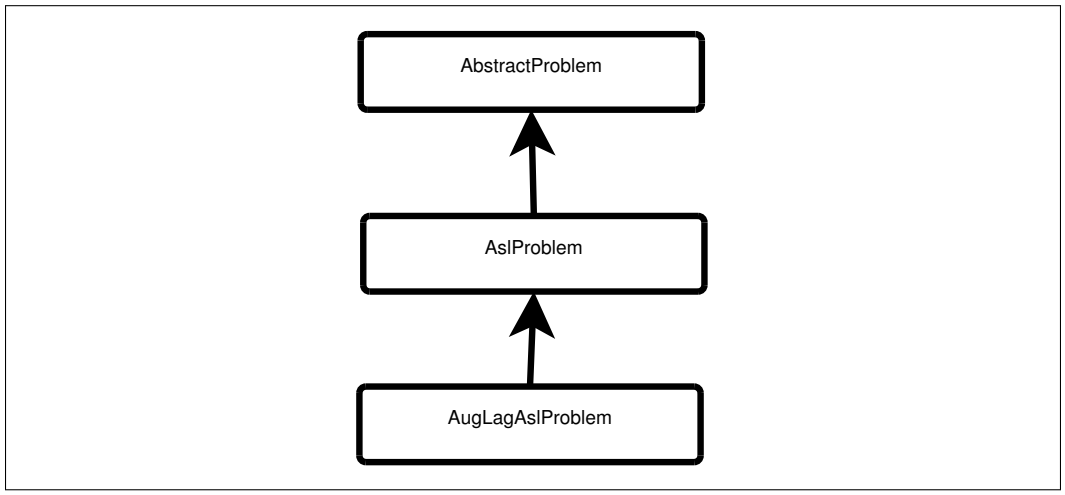


Fig. 3.4: Inheritance of FOAAL's problem classes

```
$ pyomo solve --solver=FOAAL PyomoModel.py
```

Fig. 3.5: PYOMO solve command, where solver is FOAAL.

class interfaces FOAAL with many popular modeling languages. For example FOAAL can be easily called from PYOMO [14] with the command in figure 3.5.

`AugLagAslProblem` inherits most of the `AslProblem` routines, except for the objective function value and objective function gradient computations since those need to be modified to accommodate the subproblem solver. As described in section 2 the subproblems constitute unconstrained minimizations of the form (2.3). Therefore the subproblems work with the augmented Lagrangian function instead of the actual objective function. The difference is shown in figures 3.6 and 3.7. If (2.2) contains explicit variable bounds of the form $a \leq x \leq b$, we may elect to directly enforce these in the supbroblems, rather than attaching Lagrange multipliers to them. In that case, the subproblems become box-constrained minimization problems and the gradient condition in (2.8) becomes a more complicated subgradient condition.

For parallel applications, where the objective function is known in advance, and parallel function evaluation and gradient computation are possible, we can readily construct a problem class that supports parallel executions. It is the `AbstractProblem`-derived class, through the algorithm starting point vector, that determines how decision variables are distributed and possibly replicated among processors. Thus in addition to the `AbstractVector` the other fundamental building block of our object-parallel framework is the `AbstractProblem` class.

3.2.2. The LineSearchBasedMethod and Its Derived Methods. The computationally most intensive part of FOAAL is solving the subproblems (2.3) generated by the algorithm. Therefore it is important to have infrastructure in place that allows the efficient implementation of the subproblem solvers.

```
double AslProblem::objVal(VectorObject &y) {
    numObjEval++;
    return objval(0, y.getData(), NULL);
}

void AslProblem::objGrad(VectorObject& y, VectorObject& grad) {
    numGradEval++;
    objgrd(0, y.getData(), grad.getData(), NULL);
}
```

Fig. 3.6: `AslProblem` objective value and gradient functions.

So far we have implemented two subproblem solvers in our object-parallel framework, one is a conjugate gradient (CG) method by Hager and Zhang [11, 12] for unconstrained problems, the other is a non-monotone spectral projected gradient [2, 3] (SPG) method capable of handling simple variable bounds. Both of these algorithms employ line-search procedures. Essentially, after determining a step direction d^k, they perform some kind of backtracking procedure to determine the step-size α_k in the step calculation $x^{k+1} = x^k + \alpha_k d^k$. To promote efficient implementation of such procedures, our framework provides a base class called `LineSearchBasedMethod`. This class also provides built-in members for scalar and vector quantities typically maintained by line search algorithms, including the objective values, the current iterate, the objective function gradient, and the search direction. Crucially, the `LineSearchBasedMethod` class also holds an array of `PointMemory` objects. `PointMemory` is a class designed to cache function values, gradient values, and, through derived classes, related application-specific information. `LineSearchBasedMethod` provides methods `Phi` and `GradPhi` for computing the value and gradient of the line-search function $\phi_k(\alpha) = f(x^k + \alpha d^k)$, where f is the problem objective function. In some line-search procedures, $\phi_k(\alpha)$ or $\nabla\phi_k(\alpha)$ may be evaluated more than once at the same value of α; for example, this phenomenon can occur quite often in the conjugate gradient algorithm of [13]. The caching mechanism built into the `Phi` and `GradPhi` methods prevents potentially time-consuming recomputation of the function value or gradient in such cases, while keeping the solution algorithm code free of clutter from caching-related details. Another possible situation is that the line-search algorithm may compute $\phi_k(\alpha)$ and subsequently compute $\nabla\phi_k(\alpha)$ for the same value of α. For many problem classes, these two computations share significant common underlying computations, whose results it is more efficient to cache than to recompute. Application-specific classes derived from `PointMemory` may be used for this purpose. Since most first-order methods employ some sort of line-search procedure we anticipate that these utilities will be useful for future work as well.

```
double AugLagAslProblem::objVal(VectorObject &y) {
  numObjEval++;

  double value = objval(0, y.getData(), NULL);

  if(numEquCon > 0) {
      equConVal(y, hx);
      value += lambda.inner(hx) + 0.5 * c * hx.norm2sq();;
  }

  if(numInequCon > 0) {
      inequConVal(y, gx);
      inequPen = mu + c * gx;
      inequPen.max(inequPen, zeroVec);
      value += 1.0/c * 0.5 * (inequPen.norm2sq() - mu.norm2sq());
  }
  return value;
}

void AugLagAslProblem::objGrad(VectorObject& y, VectorObject& grad) {
  numGradEval++;

  objgrd(0, y.getData(), grad.getData(), NULL);

  if(numEquCon > 0) {
      for(int i=0; i<numEquCon; ++i) {
          equConiGrad(i, y, conGradient);
          grad += (lambda[i] + c*equConiVal(i, y)) * conGradient;
      }
  }

  if(numInequCon > 0) {
      for(int i=0; i<numInequCon; ++i) {
          inequConiGrad(i, y, conGradient);
          grad += std::max( 0.0,
                            mu[i] + c * inequConiVal(i, y) )
                * conGradient;
      }
  }
}
```

Fig. 3.7: `AugLagAslProblem` objective value and gradient functions.

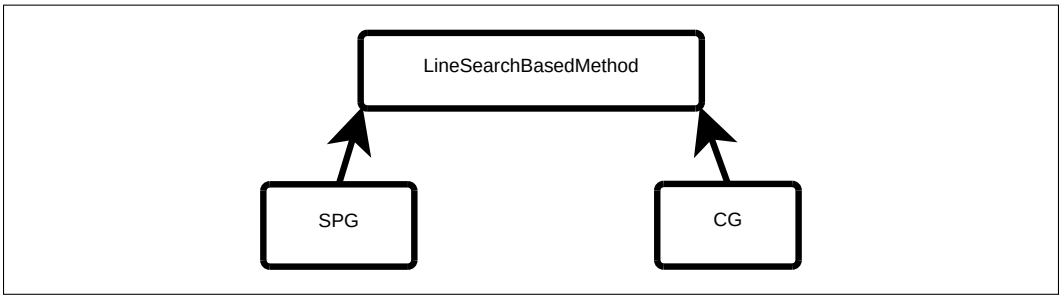


Fig. 3.8: Inheritance of FOAAL's line-search based methods

4. Solving problems with FOAAL. So far we have successfully solved smaller nonlinear programming problems with FOAAL such as: `airport`, `avgasa`, `cb3`, `discs`, `goffin`, `hadamard`, `himmelp6`, `hs117`, `hs268`, `ssebnln`, `womflet` from the CUTE test set [4], and the extensive form of a stochastic programming problem by Birge and Louveaux also known as the farmer problem [1]. Our main interest will be to solve stochastic programming problems so we will focus our attention on these problems in the following sections.

4.1. Stochastic Programming and the Unit Commitment Problem. A stochastic programming problem is a mathematical programming problem, where some of the parameters are random. We talk about solving a stochastic programming problem if we take into account the probability distribution of the random elements in the underlying problem. Stochastic programming problems are dealt with extensively in the following literature: [1, 17, 19]. We are interested in a special class of stochastic programming problems called the Unit Commitment problem.

4.1.1. The Unit Commitment Problem. The traditional Security Constrained Unit Commitment (SCUC) problem schedules the on/off states of generating units for the next operating day to minimize total production costs based on load and non-dispatchable generation forecasts subject to power balance, power transfer limits and other operational and economical constraints. This can be formulated as:

$$\begin{aligned} \min \quad & \sum_{k\in K}\sum_{j\in J} c_j^P(k) + c_j^u(k) + c_j^d(k) \\ \text{ST} \quad & \sum_{j\in J} p_j(k) = D(k), \quad \forall k \in K \\ & p_j(k) \in \Pi, \quad \forall j \in J,\ \forall k \in K \end{aligned} \tag{4.1}$$

where c_j^P represents production costs, c_j^u represents startup costs, c_j^d represents shutdown costs, $D(k)$ stands for various loads (demands) in period k and Π represents the region of feasible production of all generating units in all time periods. The specific nature of Π is model-dependent. The sets K and J are time intervals in the problem time span and the generation units in the system, respectively. Detailed development of the above model and numerous other SCUC formulation can be found in the literature [5].

In (4.1) the $D(k)$s are random and depend on consumer behavior and various other circumstances. For example, during the summer on a hot day consumers will most likely use more electricity to power air conditioning systems than on a cooler day. As a consequence we may identify three possible outcomes: high, medium and low demands and associate certain probabilities with them. This model gives us three different scenarios for our stochastic programming problem. This formulation will allow us to come up with a scenario tree that describes our optimization problem, where the possible scenarios at time period k depend on the events that took place at time periods $k-1, k-2, \ldots, 0$. If we want to solve the SCUC as a stochastic programming problem we will have to take the above mentioned scenarios and time periods into account when performing our optimization.

4.2. The serial solution of the Unit Commitment problem via FOAAL. What is most important for FOAAL is that the (4.1) formulation of the SCUC can be expressed in the form of (2.2). We will use PYOMO to generate the extensive form of the unit commitment problem using the convert option. The command is illustrated in figure 4.1; it generates a ".nl" file, which then can be handed to FOAAL. As explained in section 3.2.1 the `AugLagAslProblem` class creates a problem instance for FOAAL from the ".nl" file.

```
$ pyomo convert --output=Ef.nl RefModel.py Scenario1.dat Scenario2.dat ...
```

Fig. 4.1: PYOMO command to generate the extensive form of stochastic programming problems.

4.3. How to extend the serial Unit Commitment problem solver via FOAAL to obtain a parallel unit commitment solver. The parallel solution can be envisioned in the following manner:

1. Each processing unit receives a ".nl" file generated by PYOMO that holds information about a subset of the scenario tree called a "bundle"; see figure 4.2.
2. Variables and constraints except the ones in the leaf nodes are replicated in each processor.
3. Each processing unit then solves the given "bundle", which we'll call the scenario subproblem.
4. Each processing unit updates the Lagrange multipliers by communicating with other processes.
5. 3-4 repeats until an implementable solution is found.

In the above description the first two steps would be achieved by PYOMO, where PYOMO replicates the "non-anticipative variables" and creates the .nl files accordingly. Once a solution is found, PYOMO will have a routine in place that can merge the ".sol" files created by FOAAL.

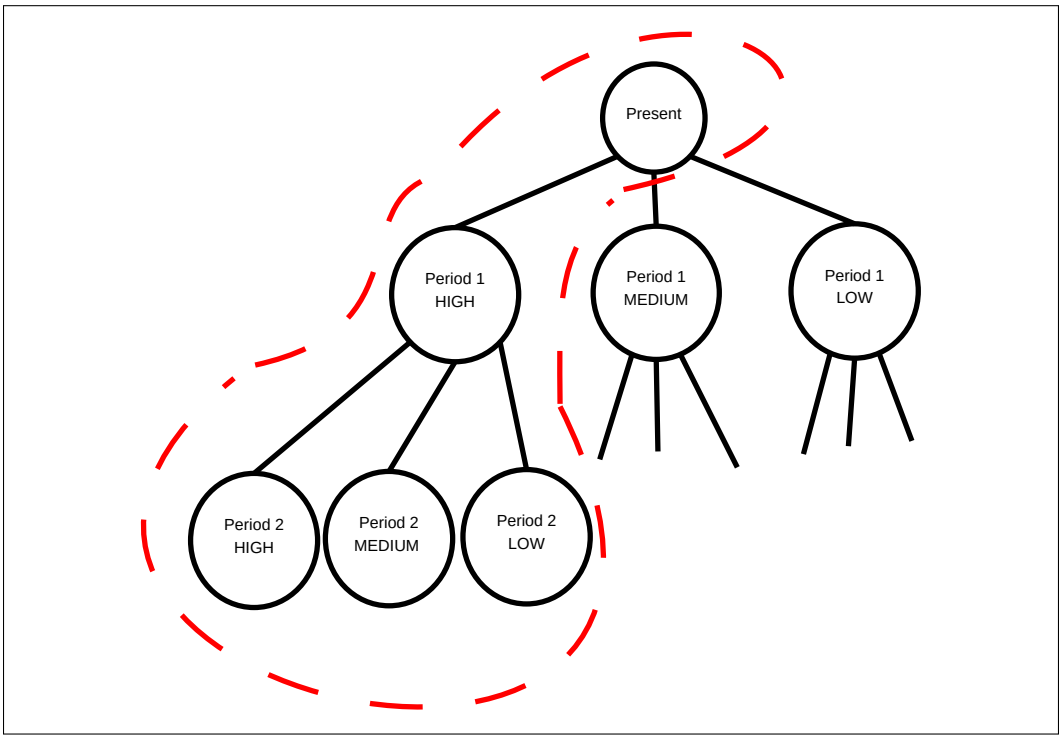


Fig. 4.2: A bundle of the scenario is tree signaled by dashed lines.

5. Conclusions. The implementation of FOAAL suggests that our object-parallel framework is suitable to implement optimization algorithms efficiently and in a readable manner. Since the resulting source-code is clean from "clutter" we assume that it will be much easier to maintain and further improve FOAAL's code-base. We hope that this will be an attractive trait of our object-parallel approach. Our philosophy is that if the application developers can focus on efficiently performing just a few operations, namely function evaluation and gradient evaluation, in whatever way they see fit, then our object-oriented framework can use those low-level operations to construct an efficient parallel solver. The benefits of the operator overloading techniques are to make the solver-level code easier to understand and maintain, and to facilitate relatively easy development of new solver algorithms as necessary.

There is still work that needs to be done for FOAAL to deliver a solution to the SCUC that is competitive with currently available solver packages. For example, it is important that FOAAL's subproblem solver handles the functions produced by the augmented Lagrangian robustly. In our experience this is not always the case with CG and SPG. However, we have also observed that if the subproblems are handled efficiently FOAAL very rarely takes more than 10 iterations to converge.

Since this project is still in its early stages we expect that these initial hurdles can be overcome and that we can use FOAAL in conjunction with PYOMO to solve real-world unit commitment problems.

REFERENCES

[1] J. Birge and F. Louveaux, *Introduction to Stochastic Programming*, Springer, 1997.
[2] E. G. Birgin, J. M. Martínez, and M. Raydan, *Nonmonotone spectral projected gradient methods on convex sets*, SIAM J. Optim., 10 (2000), pp. 1196–1211.
[3] ———, *Inexact spectral projected gradient methods on convex sets*, IMA J. Numer. Anal., 23 (2003), pp. 539–559.
[4] I. Bongartz, A. R. Conn, N. Gould, and P. L. Toint, *Cute: Constrained and unconstrained testing environment*, ACM Trans. Math. Softw., 21 (1995), pp. 123–160.
[5] M. Carrion and J. M. Arroyo, *A computationally efficient mixed-integer linear formulation for the thermal unit commitment problem*, IEEE Transactions on Power Systems, 21 (2006), pp. 1371–1378.
[6] J. Eckstein and G. Mátyásfalvi, *Object-parallel infrastructure for implementing first-order methods, with an example application to lasso*, Optimization Online: 4748, (2015).
[7] J. Eckstein and P. J. S. Silva, *A practical relative error criterion for augmented Lagrangians*, Math. Program., 141 (2013), pp. 319–348.
[8] R. Fourer, D. M. Gay, and B. W. Kernighan, *AMPL: A Modeling Language for Mathematical Programming*, Duxbury Press/Wadsworth, 1993.
[9] D. M. Gay, *Hooking your solver to ampl*, Tech. Rep. 97-4-06, Bell Laboratories, 04 1997.
[10] ———, *Writing .nl files*, Tech. Rep. SAND2005-7907P, Sandia National Laboratories, 2005.
[11] W. W. Hager and H. Zhang, *A new conjugate gradient method with guaranteed descent and an efficient line search*, SIAM J. Optim., 16 (2005), pp. 170–192.
[12] ———, *Algorithm 851: CG_DESCENT, a conjugate gradient method with guaranteed descent*, ACM Trans. Math. Softw., 32 (2006), pp. 113–137.
[13] ———, *A new active set algorithm for box constrained optimization*, SIAM J. Optim., 17 (2006), pp. 526–557.
[14] W. E. Hart, C. Laird, J.-P. Watson, and D. L. Woodruff, *Pyomo–optimization modeling in python*, vol. 67, Springer Science & Business Media, 2012.
[15] M. A. Heroux, R. A. Bartlett, V. E. Howle, R. J. Hoekstra, J. J. Hu, T. G. Kolda, R. B. Lehoucq, K. R. Long, R. P. Pawlowski, E. T. Phipps, A. G. Salinger, H. K. Thornquist, R. S. Tuminaro, J. M. Willenbring, A. Williams, and K. S. Stanley, *An overview of the Trilinos project*, ACM Trans. Math. Softw., 31 (2005), pp. 397–423.
[16] J. Ostrowski, M. F. Anjos, and A. Vannelli, *Tight mixed integer linear programming formulations for the unit commitment problem*, IEEE Transactions on Power Systems, 27 (2012), pp. 39–46.
[17] A. Prékopa, *Stochastic Programming*, Kluwer Academic Publishers, 1995.
[18] R. Rockafellar, *Augmented lagrangians and applications of the proximal point algorithm in convex programming*, Math. Oper. Res., 1 (1976), pp. 97–116.
[19] A. Shapiro, D. Dentcheva, and A. Ruszczyński, *Lectures on Stochastic Programming Modeling and Theory*, MPS-SIAM Series on Optimization, 2nd ed., 2014.
[20] B. Stroustrup, *The C++ Programming Language*, Addison-Wesley, Boston, MA, USA, 3rd ed., 2000.

LOCALIZED OPTIMIZATION-BASED REMAP FOR TRANSPORT

SCOTT A. MOE*, PAVEL B. BOCHEV†, KARA J. PETERSON‡, AND DENIS RIDZAL§

Abstract. We present a new localized conservative and bounds preserving optimization-based remap (ℓOBR) algorithm. The algorithm continues the developments of [4], which uses optimization to enforce relevant physical properties in a mass-density remap. The resulting quadratic program (QP) formulation of remap enforces mass conservation by a single linear constraint. This non-local constraint can lead to small, yet global, "mass spreading" which is undesirable in applications such as incremental remap and semi-Lagrangian transport schemes. In this context, the solution at a point in space should not be affected by information beyond a certain domain of influence determined by the time step and the specific velocity field. This motivates a modification of the OBR in which the global mass conservation constraint is no longer enforced directly. Instead global mass conservation is enforced using a two step procedure designed to minimize the subset of the grid on which the update procedure will have a nonphysical domain of dependence. In order to assign degrees of freedom to this sub-domain we will use an indicator correlated with the expected accuracy of the target solution. Since we cannot know this sub-domain analytically we propose a simple optimization procedure which uses a bisection algorithm to find a quasi-optimal approximation of this domain while ensuring that the OBR feasible set remains nonempty. Application to semi-Lagrangian transport illustrates the improvements enabled by the new ℓOBR.

1. Introduction. Preservation of relevant physical properties remains a fundamental challenge in the discretization of PDEs. This is especially true for the numerical solution of transport equations, where one has to account for properties such as conservation of mass and local solution bounds. Schemes that do not preserve these properties lead to nonphysical solutions exhibiting spurious oscillations and/or inaccurate mass and stability problems.

This work continues the development of a *divide-and-conquer* optimization-based approach for the formulation of *feature preserving* discretizations that seek to directly preserve a PDE's physical properties[7, 6, 3, 5, 4].

In [4, 2] the authors apply the *divide-and-conquer* strategy to semi-Lagrangian transport using finite volume and spectral element discretizations. This approach, termed optimization based transport (OBT), combines a Lagrangian update (discussed in the next section) step with an optimization-based remap (OBR) to transfer data between a pair of meshes one of which is static and one of which is deformed by the velocity field. In this context OBR is used to enforce mass conservation and an approximation of the monotonicity preserving property. Specifically, solution of a quadratic program (QP) that seeks to minimize the ℓ_2 norm difference between a solution satisfying these constraints and a target solution accomplishes the data transfer.

In providing an efficient way to enforce local bounds and mass conservation regardless of the underling discretization OBT has proven to be a powerful idea. Since the OBT separates accuracy and efficiency considerations from the enforcement of the physical properties, one can select the implementation of the Lagrangian update that is most appropriate for a particular application.

For example consider OBT in which the Lagrangian update is implemented by a high order spectral element discretization. With OBR it is possible to create a conservative and quasimonotone semi-Lagrangian scheme of arbitrary order that uses only interpolations in the Lagrangian update step. This scheme is extremely efficient because interpolation is a cheap operation and semi-Lagrangian schemes can avoid the reduced time-step restrictions that are usually associated with high order schemes applied to hyperbolic PDEs[2]. High

*University of Washington Department of Applied Mathematics, smoe@uw.edu
†Sandia National Laboratories, pbboche@sandia.gov
‡Sandia National Laboratories, kjpeter@sandia.gov
§Sandia National Laboratories, dridzal@sandia.gov

order semi-Lagrangian schemes that are conservative are typically much more expensive because they must proceed in a dimensional split manner and high-order splitting schemes require many sub-steps [11].

2. Mathematical Preliminaries. In this work we consider the linear transport equation

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\boldsymbol{\nu}\rho) = 0 \tag{2.1}$$

where ρ is a non-negative density function and $\boldsymbol{\nu}$ is a given velocity field. For simplicity we assume that $\nabla \cdot \boldsymbol{\nu} = 0$[1].

A semi-Lagrangian scheme is defined as any scheme that combines a Lagrangian update to advance data in time with a constrained interpolation (remap) between a pair of grids. One of these grids is a static one which contains the solution at the present and future time steps, while the second grid is an auxiliary grid obtained by projecting the fixed grid forward or backward in time; see Fig.2.1.

A Lagrangian update mathematically represents an analytical implementation of Equation (2.1) and as such it exactly preserves all of the correct physical properties. As a result, the origin of all nonphysical properties in any semi-Lagrangian scheme is confined to the remap step.

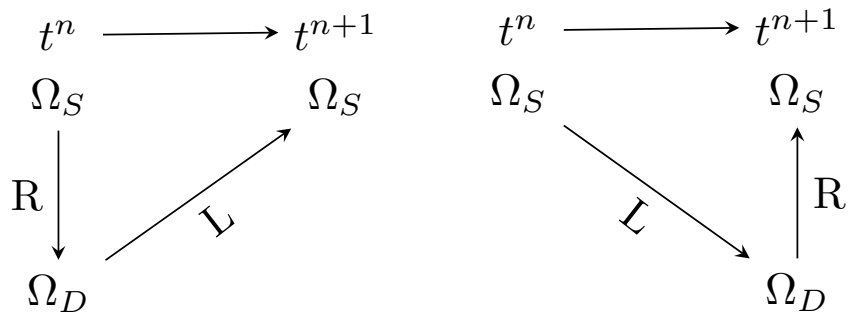
Semi-Lagrangian schemes can involve Lagrangian steps that advance forward in time or backward in time. The semi-Lagrangian algorithm, in either case, will follow one of the flow diagrams indicated in Figure 2.1. In the forward in time case data on a domain Ω_S is advected along characteristics to obtain data at some later time t^{n+1} on a deformed domain Ω_D. This data must then be remapped back onto the domain Ω_S. In the backward case the static grid Ω_S is projected backward in time along characteristics to define the deformed grid Ω_D. The data at time-step t^n is then remapped onto Ω_D. Following this data at time t^n on Ω_D is trivially advected forward in time onto the grid Ω_S at time t^{n+1}. Because the backward scheme remaps from a fixed to a deformed grid, it is preferred in settings where the former can be chosen to be uniform. In particular, it is used exclusively for tracer transport in atmospheric simulations. For this reason in this paper we focus on backward in time semi-Lagrangian schemes that use the OBR to implement the remap step. We refer to such schemes as Optimization Based Transport (OBT). An OBT scheme enforces mass conservation by a single global linear constraint, which ties together all degrees of freedom. Thus, in principle, to satisfy the mass conservation OBT may borrow from or send mass to cells beyond a given cell's domain of influence as determined by the velocity $\boldsymbol{\nu}$ and the time step. We will refer to this phenomenon as "mass-spreading".

Mass-spreading is nonphysical for transport because the characteristics should determine a finite domain of influence for every point in Ω_S. However, the global connection between all degrees of freedom can allow density to spread further than it would spread physically. This problem has previously been pointed out in the literature [1].

Remark 1. *The flux-form of OBR [5] enforces mass conservation by using conservative fluxes to compute new cell masses. This form of OBR is closely related to the Flux Corrected Remap (FCR) [10] and avoids the issue of mass-spreading. However, in the context of transport applications the flux form tends to be less efficient than the mass-form OBR.*

Remark 2. *Because at the remap step OBT finds a globally optimal solution of a QP it produces the best possible, with respect to the ℓ_2 distance, approximation of the data that also satisfies the physical bounds. In other words, a solution obtained by any other means, e.g.,*

[1]Note that this means that ρ is a both a density function and a passive tracer that is constant along characteristics.



(a) Backward semi-Lagrangian scheme (b) Forward semi-Lagrangian scheme

Fig. 2.1: The two-possible progressions of the semi-Lagrangian OBT scheme. L indicates the Lagrangian update and R indicates the constrained remap (OBR).

monotone reconstruction, or flux corrected transport can only match the unlimited solution as closely as the OBT solution and in most cases it will match less well (note that this does not guarantee that the OBT solution is the best approximation of the exact solution).

2.1. Optimization Based Remap. A key step in a backward OBT scheme is the optimization-based remap from Ω_S to Ω_D. The input for this step is the vector $\widetilde{\boldsymbol{\rho}} \in \mathbb{R}^N$ of approximate density values and vectors $\bar{\boldsymbol{\rho}}, \underline{\boldsymbol{\rho}} \in \mathbb{R}^N$ of physically-motivated local bounds for the density. In addition, we are also given a constant scalar quantity M, which specifies the total mass in Ω_S.

Remark 3. *In a finite volume scheme $\widetilde{\boldsymbol{\rho}}$ contains an approximation of the mean cell density on Ω_D computed using the solution on Ω_S. In a spectral element scheme $\widetilde{\boldsymbol{\rho}}$ contains point density values at the* departure *Gauss-Legendre-Lobatto (GLL) points in Ω_D. However OBR is completely agnostic to how $\widetilde{\boldsymbol{\rho}}$ has been obtained or what its degrees of freedom represent. The only aspect of the scheme that reflects the underlying discretization is a vector of weights $\boldsymbol{\omega} \in \mathbb{R}^N$ that enters the computation of the total mass M. In a finite volume scheme this vector represents the cell volumes, while in the spectral element case it contains the GLL quadrature weights. In what follows we will assume that OBR is being implemented using a weighted ℓ_2 norm with diagonal weight matrix $\mathbf{W}$For examples of specific implementations see [4, 8, 1, 2].*

OBT uses $\widetilde{\boldsymbol{\rho}}$, $\bar{\boldsymbol{\rho}}, \underline{\boldsymbol{\rho}}$ and M to compute a solution $\boldsymbol{\rho}^*$ on Ω_D by solving the following QP:

$$\boldsymbol{\rho}^* = \arg\min_{\boldsymbol{\rho}} \|\widetilde{\boldsymbol{\rho}} - \boldsymbol{\rho}\|_{\mathbf{W}}^2 \quad \text{such that} \quad \begin{cases} \boldsymbol{\omega}^T \boldsymbol{\rho} = M \\ \underline{\boldsymbol{\rho}}_i \leq \boldsymbol{\rho}_i \leq \bar{\boldsymbol{\rho}}_i \quad \forall 1 \leq i \leq N \end{cases} \tag{2.2}$$

This QP has an attractive structure that lends itself to efficient optimization algorithms. The inequality constraints in (2.2) enforce the local solution bounds that hold for (2.1), while the equality constraint enforces the conservation of total mass.

3. Mass Spreading with OBR. Although (2.2) can be solved very efficiently the globality of the mass conservation constraint represents a potential liability for transport applications. The reason is that solutions of (2.1) possess a finite domain of influence, i.e.,

a solution value at a given point depends only on the solution values along the streamline connecting that point to another point as determined by the velocity field and the time-step. At the same time, due to its global nature, the first constraint in (2.2) can potentially violate this property, i.e., a solution value at a given point may depend on all other solution values. This non-locality of the numerical method may result in a solution that displays nonphysical behavior.

In the following example, which is based on the example in [1] on page 268, we will illustrate non-local mass spreading with OBR. The numerical scheme implemented is a spectral element discretization using third order polynomials with semi-Lagrangian time-stepping. More information about this specific implementation of OBT can be found in [2].

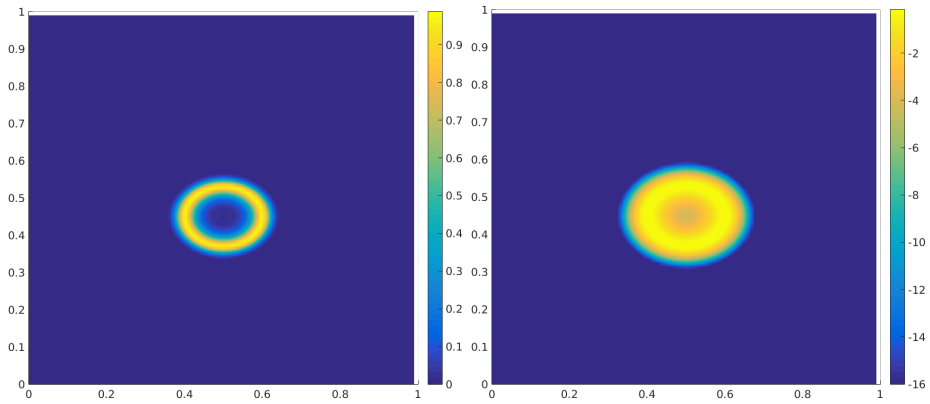
The example involves a rapidly decaying smooth density profile in a velocity field that induces solid body rotation:

$$\rho = \exp(-\beta^2((\frac{x - x_0}{0.1})^2 + (\frac{y - y_0}{0.08}) - 1)^2) \tag{3.1}$$

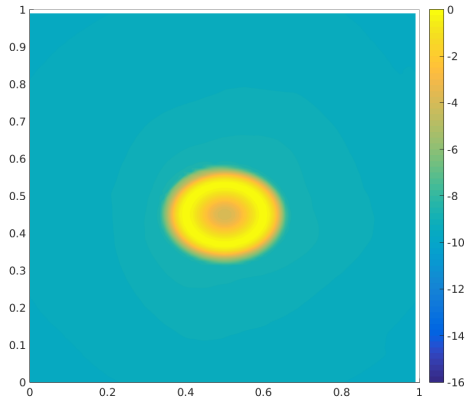
with $\beta = 2$,$x_0 = 0.5$ and $y_0 = 0.45$.

$$(u, v) = \left((y - \frac{1}{2}), -(x - \frac{1}{2})\right) \tag{3.2}$$

In Figure 3.1 we see that initially the solution sits in a zero density vacuum and since the example uses a solid body rotation velocity field one would expect the solution to still be in a vacuum at the final time. Instead we see that a small amount of mass has spread throughout the domain. The magnitude of the spreading is small so it should not be a problem in many situations, but this could be important when simulating something where zero density is qualitatively different from a non-zero but small density. The results in Figure 3.2 were obtained using a second order finite volume scheme with a slope limiter. The scheme was chosen to have a domain of influence that is much closer to the physical domain of influence. This scheme was implemented using the same number of degrees of freedom as the spectral element discretization. Notice that even though the solution does not appear to be as accurate as the spectral element solution it displays little mass spreading. From this experiment we can infer that if OBR was modified so that each degree of freedom maintained a smaller domain of influence this would mitigate the mass spreading.



(a) Distribution at $t = 0$ (b) Distribution at $t = 0$ (Log Scale)



(c) Distribution at $t = 2\pi$ (Log Scale)

Fig. 3.1: Solid body rotation of a smooth density profile using the semi-Lagrangian spectral element method with degree 3 polynomials and OBR based limiting[2]. These examples were run on an 80×80 mesh with 9 DOFs per cell. At the final time much of the domain has nonzero density.

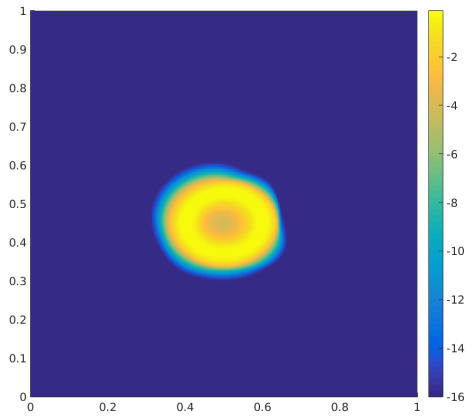


Fig. 3.2: Solid body rotation of a smooth density profile using a finite volume discretization with linear a linear reconstruction and a Van-Leer limiter. This problem was run on a 240×240 mesh.

4. A Simple Smoothness Indicator. When OBR computes a solution it is computing the closest solution in the ℓ_2 norm to the target solution. However the accuracy of the target solution itself could be highly variable throughout the domain. For example in any large regions where the solution takes a constant value the target is most likely extremely accurate (this is why mass-spreading is most visible in the large constant regions of the domain).

Perhaps one way to control mass spreading is to restrict the domain used in the optimization problem to a region where one can expect that the target solution will be least accurate. Typically the numerical approximation to the solution of a PDE will be least accurate in areas of low regularity or high variation. Assuming that the upper and lower bounds used in OBR are somewhat accurate we can use $(\bar{\rho}_i - \underline{\rho}_i)$ as an estimate of the size of the variation in the target solution near degree of freedom i. This is essentially a very simple to evaluate smoothness indicator.

5. Localized OBR. One way to reduce the non-physical spread of mass from OBR would be to split it into two optimization problems. One optimization problem would only enforce quasimonotonicity. The other optimization problem would enforce global conservation as well. Of course these problems could not be completely decoupled as the solution of the first optimization problem would change the conservation constraint in the second optimization problem.

Define the set of all cell indices $I = \{i | 1 \leq i \leq N\}$, and also define two disjoint subsets of I, I_1 and I_2, such that $I = I_1 \cup I_2$ and $I_1 \cap I_2 = \emptyset$. Let us define $\boldsymbol{\rho}_{I_2}$ and $\boldsymbol{\rho}_{I_1}$ as the portions of the vector $\boldsymbol{\rho}$ that correspond to subset I_2 or I_1 respectively. Additionally we will need to define modified weight matrices $\mathbf{W}_{I_1}$ and $\mathbf{W}_{I_2}$. Computing these two matrices is trivial as the weight matrix $\mathbf{W}$ is diagonal. We will then split the typical OBR optimization problem into two optimization problems that must be solved in the order listed:

$$\boldsymbol{\rho}^*_{I_2} = \arg\min_{\boldsymbol{\rho}} \|\widetilde{\boldsymbol{\rho}}_{I_2} - \boldsymbol{\rho}_{I_2}\|^2_{\mathbf{W}_{I_2}} \text{ s.t. } \quad \underline{\boldsymbol{\rho}}_i \leq \boldsymbol{\rho}_i \leq \bar{\boldsymbol{\rho}}_i \quad \forall i \in I_2 \tag{5.1}$$

$$\boldsymbol{\rho}^*_{I_1} = \arg\min_{\boldsymbol{\rho}} \|\widetilde{\boldsymbol{\rho}}_{I_1} - \boldsymbol{\rho}_{I_1}\|^2_{\mathbf{W}_{I_1}} \text{ s.t. } \quad \begin{cases} \boldsymbol{\omega}^T_{I_1}\boldsymbol{\rho}_{I_1} = M - \boldsymbol{\omega}^T_{I_2}\boldsymbol{\rho}^*_{I_2} \\ \underline{\boldsymbol{\rho}}_i \leq \boldsymbol{\rho}_i \leq \bar{\boldsymbol{\rho}}_i \quad \forall i \in I_1 \end{cases}$$

Using the smoothness indicator defined in Section 4 it is possible to create an algorithm that decomposes the degrees of freedom into two sets. Notice that

$$\bar{\rho}_i - \underline{\rho}_i \geq 0 \text{ and } \bar{\rho}_i - \underline{\rho}_i \leq \max_i \left(\bar{\rho}_i - \underline{\rho}_i\right)$$

If we define

$$I_c = \{i | \bar{\rho}_i - \underline{\rho}_i \geq C \max_i \left(\bar{\rho}_i - \underline{\rho}_i\right)\}$$

then $|I_{0.0}| = |I|$ and $I_{1.0}$ gives the subset of the domain that has the largest possible variation. It seems likely that the target solution is least accurate on the subset $|I_{1.0}|$. This suggests that we should attempt to find the maximum value of $C \in [0, 1]$ such that $\exists$ a solution to Equation (5.1) with $I_1 = I_c$. Equation (5.1) has a solution as long as the constraint

$$\boldsymbol{\omega}^T_{I_1}\boldsymbol{\rho}_{I_1} = M - \boldsymbol{\omega}^T_{I_2}\boldsymbol{\rho}_{I_2}$$

does not conflict with the constraint

$$\underline{\boldsymbol{\rho}}_i \leq \boldsymbol{\rho}_i \leq \bar{\boldsymbol{\rho}}_i \quad \forall i \in I_1$$

THEOREM 5.1. *The optimization problem in Equation* (5.1) *has a solution iff*

$$\boldsymbol{\omega}_{I_1}^T \underline{\boldsymbol{\rho}}_{I_1} \leq M - \boldsymbol{\omega}_{I_2}^T \boldsymbol{\rho}_{I_2}^* \leq \boldsymbol{\omega}_{I_1}^T \bar{\boldsymbol{\rho}}_{I_1} \tag{5.2}$$

Proof. The function $\boldsymbol{\omega}_{I_1}^T \underline{\boldsymbol{\rho}}_{I_1}$ is a multilinear function. If we parameterize the line between $\mathbf{x}_1 = \underline{\boldsymbol{\rho}}_{I_1}$ and $\mathbf{x}_2 = \bar{\boldsymbol{\rho}}_{I_1}$ by a single parameter θ then $\boldsymbol{\omega}_{I_1}^T \underline{\boldsymbol{\rho}}(\theta)_{I_1}$ is a continuous function of θ that obtains values on either side of

$$\mathbf{x}_3 = M - \boldsymbol{\omega}_{I_2}^T \boldsymbol{\rho}_{I_2}$$

So by the Mean Value Theorem or the properties of multilinear functions $\exists\ \theta$ such that

$$\boldsymbol{\omega}_{I_1}^T \boldsymbol{\rho}_{I_1}(\theta) = \mathbf{x}_3$$

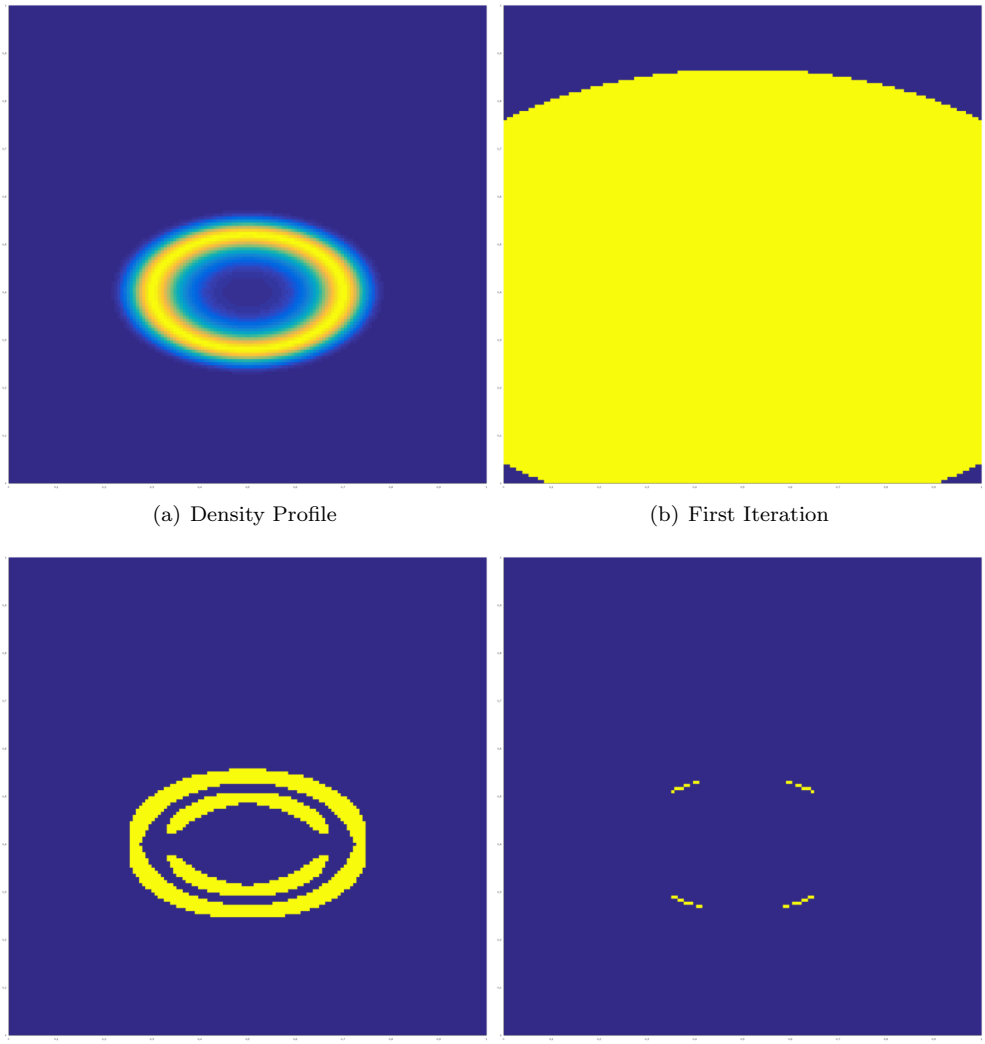
To show this from the other direction notice that if $\exists\ \theta$ such that $\boldsymbol{\omega}_{I_1}^T \boldsymbol{\rho}_{I_1}(\theta) = \mathbf{x}_3$ such that $\underline{\boldsymbol{\rho}}_i \leq \boldsymbol{\rho}(\theta)_i \leq \bar{\boldsymbol{\rho}}_i \ \ \forall i \in I_1$ then we can easily see that Equation 5.2 must be satisfied by taking the product of this inequality with $\boldsymbol{\omega}_{I_1}^T$. □

5.1. The Support Minimization Algorithm. Verifying whether the property in Equation (5.2) is satisfied only involves computing dot products and so it can rapidly be done repeatedly. This becomes the basis for an efficient algorithm to implement this localized version of OBR.

1. Initialize $C_{o1}, C_n = 0, C_{o2} = 1$, $k = 0$ and $N_1 = |I|$
2. While $k < MaxIter$
3. Compute $I_c = \{i \in I | (\bar{\rho}_i - \underline{\rho}_i) > C_n \max_i(\bar{\rho}_i - \underline{\rho}_i)\}$
4. Compute $I_2 = I \setminus I_c$
5. Find $\boldsymbol{\rho}_{I_2} = median\left(\underline{\boldsymbol{\rho}}_{I_2}, \widetilde{\boldsymbol{\rho}}_{I_2}, \bar{\boldsymbol{\rho}}_{I_2}\right)$
6. Compute $B_l = \boldsymbol{\omega}_{I_c}^T \underline{\boldsymbol{\rho}}_{I_c}$ and $B_u = \boldsymbol{\omega}_{I_c}^T \bar{\boldsymbol{\rho}}_{I_c}$
7. if $B_l \leq M - \boldsymbol{\omega}_{I_2}^T \boldsymbol{\rho}_{I_2} \leq B_u$
   - $I_1 := I_c$
   - Set $k = k + 1$, $C_{o1} = C_n$, $C_n = \frac{C_n + C_{o2}}{2}$
   - if $|I_c| = N_1$ break, otherwise set $N_1 = |I_c|$.
8. else $k = k + 1, C_{o2} = C_n, C_n = \frac{C_n + C_{o1}}{2}$

Increasing C monotonically reduces $|I_c|$ so a maximum admissible C value is guaranteed to exist. However the above bisection algorithm is not guaranteed to find the absolute minimum feasible set, instead it finds a support set that is nearly minimal. Because of this we say that the set computed is the quasiminimal feasible set. In Figures 5.1 and 5.2 we see the general performance of this algorithm. The first density profile is defined in Section 3. The second density profile originates in [9]. In each block we plot the density profile in Figure (a). In the subsequent figures we plot an indicator function that is 1 if the cell is in I_1 and 0 otherwise. The color-scale for all of those plots has 1 corresponding to yellow and 0 corresponding to dark blue. We see that for the smooth profile this algorithm creates a tiny set I_1. This indicates that the underlying approximation target (in this case obtained by a finite volume scheme) is doing very well. However on the discontinuous example we see that I_1 is somewhat larger and in the end it contains essentially the cells on which the density profile is discontinuous.

5.1.1. Computational Cost. In general predicting the computational cost of this algorithm is difficult as it relies on a number of factors such as the density profile this method is being applied to (in particular its smoothness). However we can see the scaling behavior of each step:



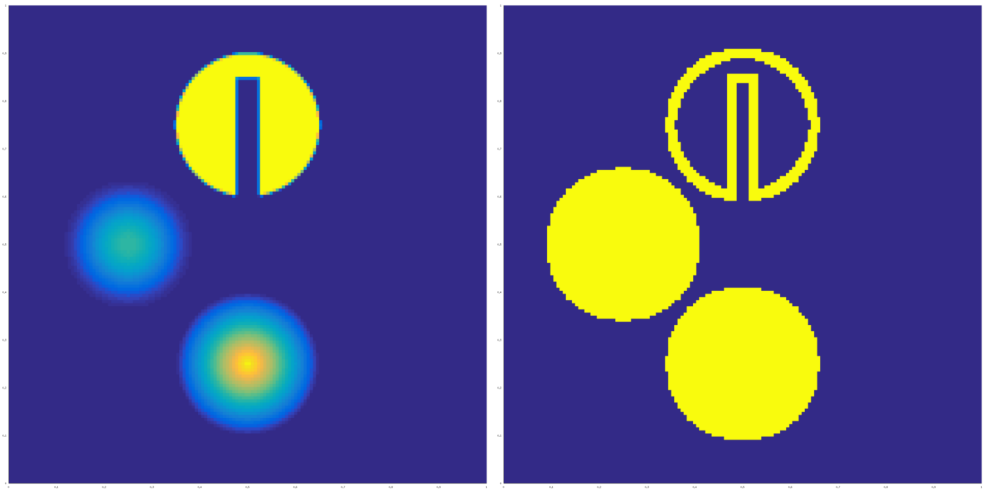
(a) Density Profile (b) First Iteration

(c) Second Iteration (d) Seventh Iteration

Fig. 5.1: The results of the algorithm described in Section 5.1 applied to the profile shown in figure (a). The other figures show the cells in set I_1 after various numbers of iterations of the algorithm introduced in Section 5.1.

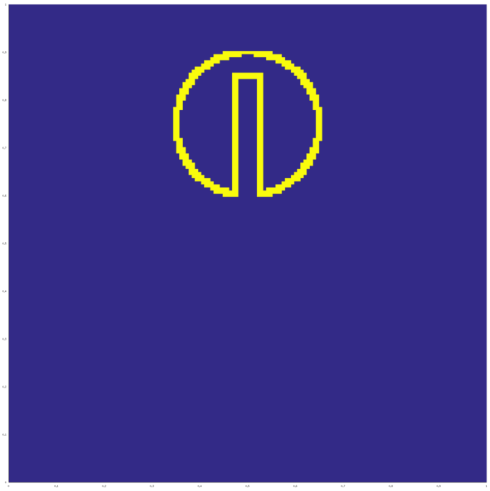
- The first search to obtain I_c will scale like N
- Finding $\boldsymbol{\rho}_{I_2}$ scales like $|I_2|$
- Testing if the proposed I_c is a feasible set will scale like $|I_c| + |I_2| = N$ (adding the costs of several vector dot products)
- The previous steps will be repeated some number of times until a good subset is discovered, but it is impossible to know how many times it must be repeated.
- Finally the constrained optimization scheme itself will scale like $|I_1|$.

No step of this algorithm scales worse than the quadratic program in global OBR. The localized OBR may even be cheaper if the globally coupled constrained optimization problem

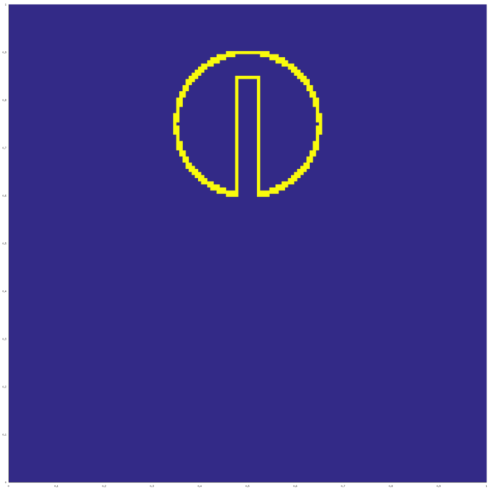


(a) Density Profile

(b) First Iteration



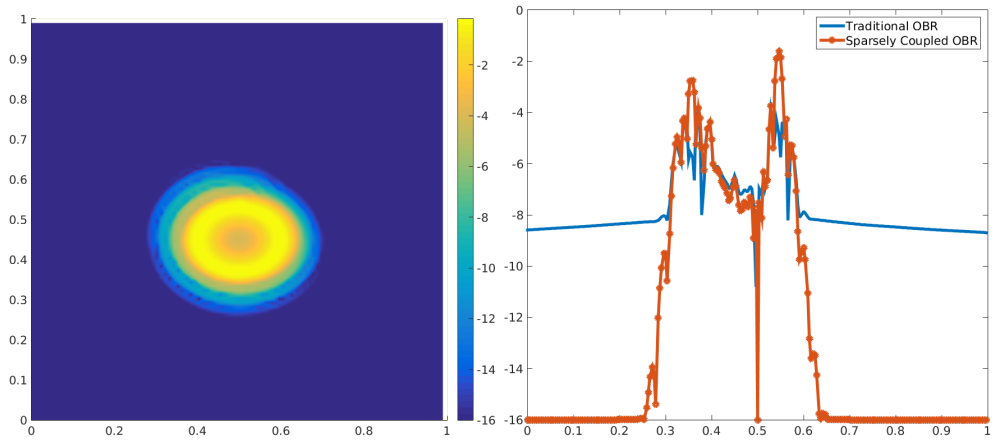
(c) Second Iteration



(d) Fourth Iteration

Fig. 5.2: The results of the algorithm described in Section 5.1 applied to the profile shown in figure (a). The other figures show the cells in set I_1 after various numbers of iterations of the algorithm introduced in Section 5.1.

is slow to converge.



(a) Final Distribution using OBR(Log Scale) (b)

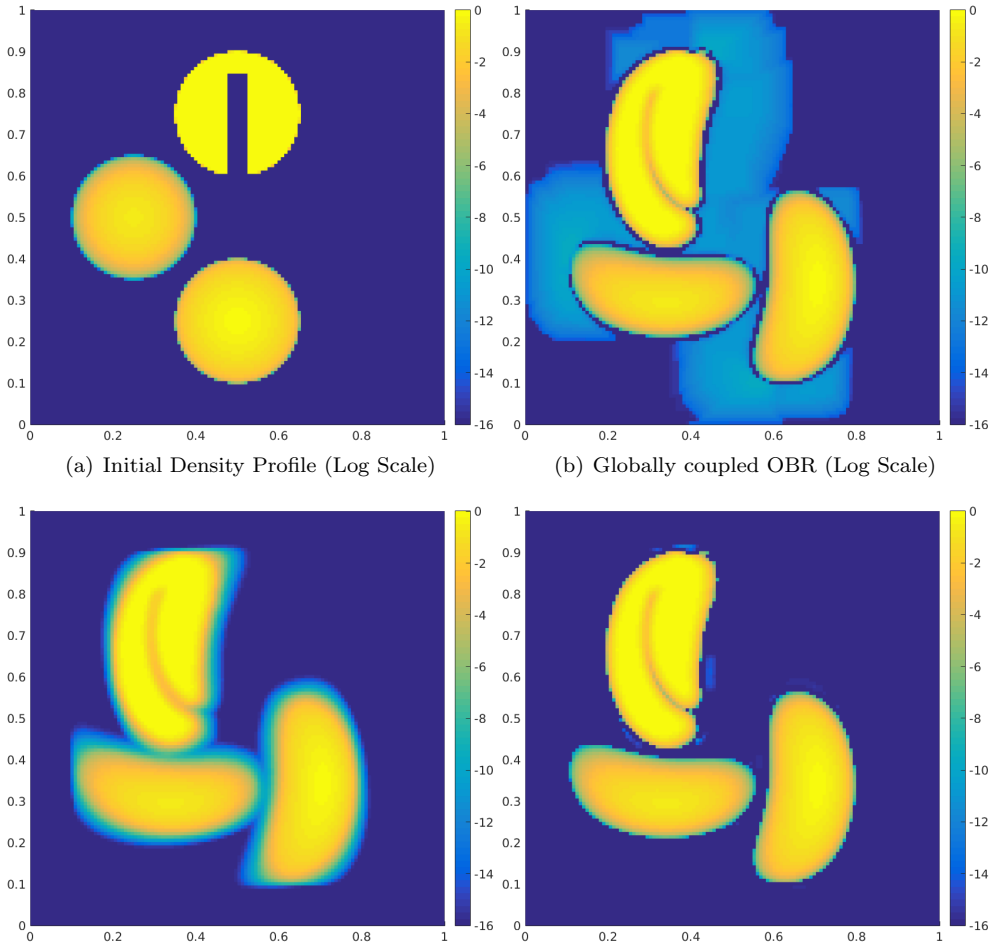
Fig. 6.1: In Figure (a) we rerun the example from Section 3 with the Sparse OBR scheme. Figure (b) shows slices at $x = 0.53$ of the errors obtained by both methods.

6. Numerical Results.

6.1. The Smooth Example. Re-implementing the example from Section 3 with the sparse OBR scheme we obtain the results shown in Figure 6.1. Compare this slice plot to the slice plot in [1]. The error for the sparse OBR scheme is machine precision outside of the center of the mesh where the density has high variation. The one negative of this scheme is that it has a relatively large point-wise error in two locations of higher variation. This is likely because for this smooth example very few cells are included in the optimization step, and so this solution cannot be as accurate as global OBR. Perhaps it would be possible to mitigate this by forcing the optimization scheme to find a partition of cells that has some higher minimum number of cells in I_1, but this would introduce a free parameter into the algorithm.

6.2. A Discontinuous Example. To illustrate the performance of this scheme on a discontinuous example we will turn to the example in Figure 5.2. This example will be run on a 125×125 degree of freedom finite volume grid using global OBR, localized OBR and linear reconstruction with Van-Leer slope limiting. We will also switch velocity fields, to show localized OBR's performance in a deformational flow. Our velocity field is given in Equation 6.1. Looking at the results we see that localized OBR appears to give the best results even beating the slope limiting scheme.

$$\begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} \sin^2(\pi x)\sin(2\pi y)g(t) \\ -\sin^2(\pi y)\sin(2\pi x)g(t) \end{bmatrix} \tag{6.1}$$



(a) Initial Density Profile (Log Scale)

(b) Globally coupled OBR (Log Scale)

(c) $t = 0.25$ Van-Leer Limiting (Log Scale)

(d) $t = 0.25$ Sparse OBR (Log Scale)

Fig. 6.2: The results of the algorithm described in Section 5.1 applied to the profile shown in figure (a). The other figures show how sparse OBR compares to OBR and a traditional slope limiting scheme.

7. Conclusions. We have introduced a localized, conservative and bounds preserving OBR scheme that effectively eliminates nonphysical mass spreading. This localized OBR computes a quasiminimal feasible set on which the conservation constraint must be enforced. The quasiminimal set is found by an efficient bisection method. This additional optimization step scales the same as the QP in the global OBR scheme so performance is not negatively impacted by switching from global OBR to localized OBR. Schemes using this localized OBR produce results that, in terms of mass spreading, are comparable to methods with numerical domains of influence that match the physical domains of influence implied by the PDE. Additionally we have effectively implemented localized OBR using both finite volume and spectral element discretizations. In fact localized OBR is identical no matter what numerical discretization it is combined with. It should be possible to extend this algorithm to other optimization based schemes because the steps to find the quasiminimal feasible set

are unaffected by the time-stepping scheme used. Future work should focus on exploring this possibility.

REFERENCES

[1] R. Anderson, V. Dobrev, T. V. Kolev, and R. Rieben, *Monotonicity in high-order curvilinear finite element arbitrary lagrangian–eulerian remap*, International Journal for Numerical Methods in Fluids, 77 (2015), pp. 249–273.

[2] P. Bochev, S. Moe, K. Peterson, and D. Ridzal, *A conservative, optimization-based semi-lagrangian spectral element method for passive tracer transport.*

[3] P. Bochev and D. Ridzal, *Additive operator decomposition and optimization–based reconnection with applications*, in Large-Scale Scientific Computing, Springer, 2010, pp. 645–652.

[4] P. Bochev, D. Ridzal, and K. Peterson, *Optimization-based remap and transport: A divide and conquer strategy for feature-preserving discretizations*, Journal of Computational Physics, 257 (2014), pp. 1113–1139.

[5] P. Bochev, D. Ridzal, G. Scovazzi, and M. Shashkov, *Constrained-optimization based data transfer*, in Flux-Corrected Transport, Springer, 2012, pp. 345–398.

[6] P. Bochev, D. Ridzal, and J. Young, *Optimization–based modeling with applications to transport: Part 1. abstract formulation*, in Large-Scale Scientific Computing, Springer, 2012, pp. 63–71.

[7] P. B. Bochev and D. Ridzal, *An optimization-based approach for the design of pde solution algorithms*, SIAM Journal on Numerical Analysis, 47 (2009), pp. 3938–3955.

[8] O. Guba, M. Taylor, and A. St-Cyr, *Optimization-based limiters for the spectral element method*, Journal of Computational Physics, 267 (2014), pp. 176–195.

[9] R. J. LeVeque, *High-resolution conservative algorithms for advection in incompressible flow*, SIAM Journal on Numerical Analysis, 33 (1996), pp. 627–665.

[10] R. Liska, M. Shashkov, P. Váchal, and B. Wendroff, *Optimization-based synchronized flux-corrected conservative interpolation (remapping) of mass and momentum for arbitrary Lagrangian-Eulerian methods*, J. Comput. Phys., 229 (2010), pp. 1467–1497.

[11] J. A. Rossmanith and D. C. Seal, *A positivity-preserving high-order semi-lagrangian discontinuous galerkin scheme for the vlasov–poisson equations*, Journal of Computational Physics, 230 (2011), pp. 6203–6232.

A MODIFICATION TO THE REMAPPING OF GAUSS-LOBATTO NODES TO THE CUBED SPHERE

MIRANDA R. MUNDT[†], MARK B. BOSLOUGH[‡], MARK A. TAYLOR[§], AND ERIKA L. ROESLER[¶]

Abstract. The implementation of climate models is essential to studying the changes in our world. A crucial piece of this model is the dual grid, needed for use of the High-Order Methods Modeling Environment (HOMME). This paper details the efforts to create a more accurate dual grid, and then analyzes the efficiency of said dual grid when used with multiple remapping algorithms.

1. Introduction. Atmosphere, ocean, and land surface models require detailed and specific methods. One of the essential pieces of HOMME (High-Order Methods Modeling Environment) is the dual grid, a mesh which contributes to the conservation of water in climate simulations. The successful creation and implementation of this grid has been a focus point of research, as adequate dual grids improve the conservation[1] of flux fields, so as to preserve energy and water when interpolating between meshes, for example.

Definition 1. *A **grid** in this context is defined as a pattern of lines on the sphere, such as a latitude-longitude split of the globe.*

Definition 2. *A **map** is defined as a mathematical function with a specified structure.*

Definition 3. ***Remapping** is the process of applying a mathematical function to existing data to interpolate the information from one grid type (i.e., cubed-sphere) to another grid type (i.e., latitude-longitude).*

In this paper, we detail the efforts to create a satisfactory dual grid. Section 2 describes our first step in achieving this goal, which was to alter the existing mathematical map by [1] to adjust for both global and local surface area discrepancies which arise from too low accuracy. Section 3 details our work on creation of the dual grid, including information on the iterative method conceived and the results produced. Finally, Section 4 details our analysis of the resulting dual grid using utilities provided by two services: those supplied by the Earth System Modeling Framework (ESMF), a well-established open-source software package[2], and TempestRemap, a new software package by Paul Ullrich[3] of the University of California, Davis, whose development is ongoing.

2. Mapping. There are multiple mesh types available on which to conduct climate simulations, such as latitude-longitude, cubed-sphere, and geodesic. Of these three meshes, we will work with the cubed sphere (Figure 2.1), for which computations are made using the spectral element method[4], which requires quadrilaterals with no hanging nodes (a node which is not a vertex of every neighboring quadrilateral)[6].

The cubed-sphere grid is generated by mapping a point on the unit square or reference element, where $U_S = \{(x_1, x_2) : -1 \leq x_1, x_2 \leq 1\}$, $S = \{(\lambda, \theta) : -\frac{\pi}{2} \leq \theta \leq \frac{\pi}{2}, 0 \leq \lambda \leq 2\pi\}$, into latitude-longitude coordinates on the unit sphere, with an intermediate mapping through the corresponding Cartesian coordinates. In a previous work, this map, denoted $\boldsymbol{r}(\boldsymbol{x}) : U \mapsto S$, was developed for each quadrilateral element Ω_m on the surface of the unit

[†]University of California, Los Angeles, mundt@ucla.edu
[‡]Multiphysics Applications, Sandia National Laboratories, mbboslo@sandia.gov
[§]Multiphysics Applications, Sandia National Laboratories, mataylo@sandia.gov
[¶]Geophysics and Atmospheric Sciences, Sandia National Laboratories, elroesl@sandia.gov

[1]See [2] for a mathematical explanation of conservation.
[2]Information and source code at https://www.earthsystemcog.org/projects/esmf/
[3]Information and source code at https://github.com/ClimateGlobalChange/tempestremap

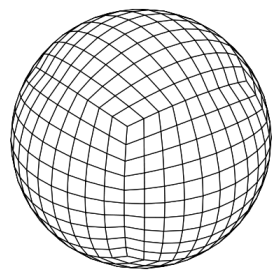


Fig. 2.1: Cubed-Sphere Mesh: 600 elements, 9° Resolution

sphere, where $\{\boldsymbol{c}_1, \boldsymbol{c}_2, \boldsymbol{c}_3, \boldsymbol{c}_4\}$ is the set of four Cartesian coordinate vectors of the vertices of Ω_m. The map is as follows ([1]):

$$\boldsymbol{r}(\boldsymbol{x}) = \frac{\tilde{\boldsymbol{r}}}{\|\tilde{\boldsymbol{r}}\|_2} \tag{2.1}$$

where

$$\tilde{\boldsymbol{r}} = \frac{1}{4}\Big[(1-x_1)(1-x_2)\boldsymbol{c}_1 + (1+x_1)(1-x_2)\boldsymbol{c}_2 + (1+x_1)(1+x_2)\boldsymbol{c}_3 + (1-x_1)(1+x_2)\boldsymbol{c}_4\Big].$$

2.1. Error Analysis. We are interested in the level of distortion and error that was present in this mapping. To investigate this, we looked closely at each individual quadrilateral's surface area and see precisely how much it varied from our expected surface area. The exact surface area of each element was calculated using l'Huiller's formula for the surface area of a spherical triangle ([8]):

$$\begin{aligned} A_m &= ER^2 \\ E &= 4\arctan\sqrt{\tan\frac{s}{2}\tan\frac{s-a}{2}\tan\frac{s-b}{2}\tan\frac{s-c}{2}} \\ s &= \frac{a+b+c}{2} \end{aligned} \tag{2.2}$$

where a, b, c are side lengths and $R = 1$ since our sphere is unit. To use this formula, we arbitrarily divide each quadrilateral into two triangles. Ultimately, we have two values E_1, E_2 which correspond to two triangular areas, and we sum these areas to get the actual area of each quadrilateral element Ω_m.

The equation for our approximated area of each quadrilateral can be found using information from equation (2.1):

$$A = \sum_{j=1}^{4}\sum_{i=1}^{4} det(D(x_i, x_j))w_i w_j \tag{2.3}$$

where $D(x_i, x_j)$ is the derivative matrix of $\boldsymbol{r}(x_i, x_j)$ and (x_i, x_j) are Gauss-Lobatto nodes with associated weights w_i, w_j. The difference between the results of equations (2.2) and (2.3) is the error $E_m = \left|\frac{A-A_m}{A_m}\right|$ present in the mesh's local surface area.

One of the goals of our analysis was to reduce the error which occurs in coarse regions of variable meshes to below 10^{-12}, which was as high as order 10^{-9} for some quadrilaterals on our test mesh, by adjusting our map to locally correct any surface area error for each individual quadrilateral. In theory, if we corrected each quadrilateral individually, this would, in turn, correct for errors in the total surface area. We detail this effort in the following subsection.

2.2. Method. To start, we edit our current map $\boldsymbol{r}(\boldsymbol{x}) : U \mapsto S$ (equation (2.1)). Our goal was to create a map which would keep the original values of $\boldsymbol{r}(\boldsymbol{x}) : U \mapsto S$ whenever $x_1 = \pm 1$ or $x_2 = \pm 1$, with the intent to maintain continuity across edges, and would adjust the area of each individual quadrilateral on the unit sphere to match its precise area, calculated exactly using l'Huiller's formula (2.2). To do this, we would need to incorporate an adjustment term ϵ per quadrilateral, which would correct the individual area errors.

We use bisection and interpolation methods in order to find the most effective ϵ for each quadrilateral. We compose the original map with a new translation map $\boldsymbol{b}(\boldsymbol{x};\epsilon) : \mathbb{R}^{2x2} \mapsto \mathbb{R}^{2x2}$, such that along the edges, $\boldsymbol{b}(\boldsymbol{x};\epsilon) = (x_1, x_2)$ to maintain the continuity constraint. This composition map would look like $\boldsymbol{r}_2 = \boldsymbol{r}(\boldsymbol{b}(\boldsymbol{x};\epsilon))$, which has a derivative matrix of the form $\frac{d\boldsymbol{r}_2}{d\boldsymbol{x}} = D(\boldsymbol{b}(\boldsymbol{x};\epsilon))\frac{d\boldsymbol{b}}{d\boldsymbol{x}}(\boldsymbol{x};\epsilon)$, where D is defined as in equation (2.3).We chose $\boldsymbol{b}$ of the form:

$$\boldsymbol{b}(\boldsymbol{x};\epsilon) = (x_1 - \epsilon(1-x_1^2)(1-x_2^2), x_2 - \epsilon(1-x_1^2)(1-x_2^2)). \tag{2.4}$$

This map fit the required constaints perfectly. It would also translate a point from the unit square into another point on the unit square for small enough epsilon, which . We then used this to create our new $\boldsymbol{r}_2(\boldsymbol{x})$ map, which would depend on $x_1, x_2, \boldsymbol{r}$, and ϵ. We also added another piece to this map, an adjustment constant α to correct for any error in the total area of the sphere from the original $\boldsymbol{r}(\boldsymbol{x}) : U \mapsto S$ map. That is:

$$\alpha = \sqrt{\frac{4\pi}{A}}$$

where A is defined as in equation (2.3). Putting it all together, our composition map became:

$$\boldsymbol{r}_2 = \alpha \boldsymbol{r}(\boldsymbol{b}(\boldsymbol{x};\epsilon)). \tag{2.5}$$

In order to solve for the unknown constant ϵ, we use MATLAB's[4] built-in function `fzero` on each quadrilateral. The function `fzero` works following the command `x = fzero(fun,x0)`, which tries to find `x` such that `fun(x) = 0` using `x0` as an initial guess.

Using this tool, we were able to combine all of the parts of the map which relied upon ϵ in order to find values which made the relative error $E_m = \left|\frac{A-A_m}{A_m}\right|$ equal to zero. The issue now, though, was that some of the values of ϵ were simply too large. For example, in our test data file[5], one quadrilateral had an ϵ value of -1.44. Physically, this would mean that $\boldsymbol{b}(\boldsymbol{x})$ was mapping a point from the unit square to a point outside of the unit square, which violates one of the requirements imposed upon $\boldsymbol{b}(\boldsymbol{x})$.

Upon further analysis, the reason for this issue was revealed. As seen in Figure 2.2, the original choice for $\boldsymbol{b}(\boldsymbol{x})$ was not a monotone function of ϵ. In fact, no matter the choice of ϵ, it was only possible to make the surface area of each quadrilateral larger, which meant that those quadrilaterals with a surface area which was approximated as greater than the

[4]MATLAB version 2012a.

[5]A plot of this is shown in Figure 2.1.

exact value could only get larger and thus further away from the exact value. This resulted in inaccurate, and relatively substantial, ϵ values.

We can see from Figures 2.2(a), (b), which correspond to the mapping from equation (2.4), that with a positive and a negative value for ϵ, the surface is still being "stretched." Alternatively, we can see in Figures 2.2(c), (d), that with a different mapping, we are able to get a surface which both shrinks and grows, depending on the choice for ϵ. The mapping used for the second figure is:

$$\boldsymbol{b}(\boldsymbol{x}) = (x_1 - \epsilon x_1(1 - x_1^2)(1 - x_2^2), x_2 - \epsilon x_2(1 - x_1^2)(1 - x_2^2)). \tag{2.6}$$

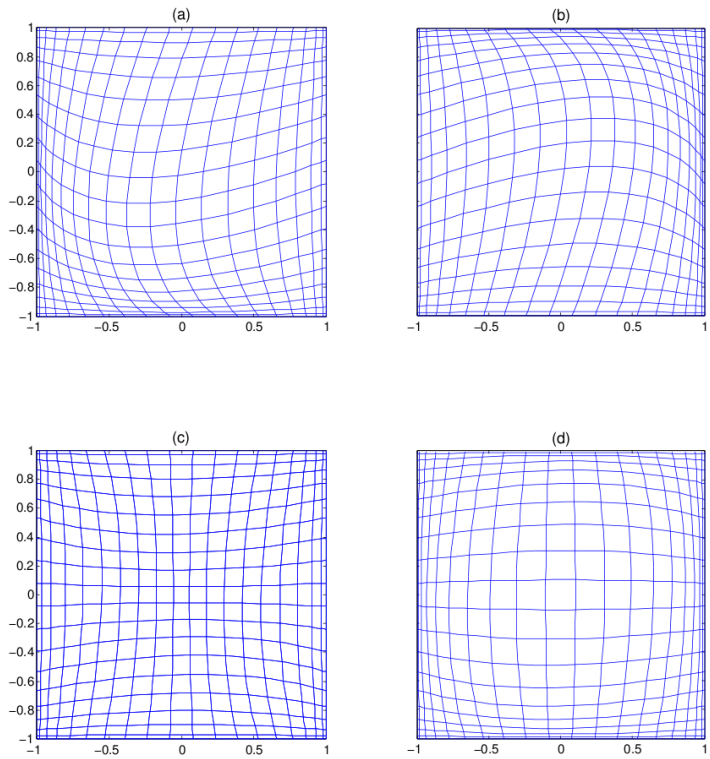


Fig. 2.2: Map Comparison: (a) Original map, $\epsilon = 0.3$; (b) Original map, $\epsilon = -0.3$; (c) Modified map, $\epsilon = 0.3$; (d) Modified map, $\epsilon = -0.3$

Here not only does this map allow for "shrinks" and "stretches", but it also follows the continuity guidelines for the edges. That is, the edges stay the same while the internal points move as necessary to adjust surface area. This map, unlike that in equation (2.4), is monotonic in ϵ, meaning that we are able to get both larger and smaller surface areas as needed. In fact, using the ϵ values generated for our test file, the error per each quadrilateral was either reduced to order 10^{-17} or vanished completely while ϵ was sufficiently small - i.e., never greater order than 10^{-3}. Thus, we could locally correct for surface area errors, meaning that our mapping from the cube to the sphere was improved greatly.

3. Dual Grid. Having established a more effective mapping method, we then moved on to the task of generating adequate dual grids for use in HOMME. An adequate dual

grid must conform with several guidelines: (1) each polygon of the dual grid must contain exactly one Gauss-Lobatto node; (2) the surface area of each polygon must equal the weight associated with the contained Gauss-Lobatto node, to a specified order of accuracy.

3.1. Method. In order to create the dual grid, we produced an iterative method. This method, coded and implemented using MATLAB, changes the surface area of all polygons simultaneously by a slight amount with each iteration in two iterative stages. In an initialization stage, the parameters for the iteration are defined (see Table 3.1). As we can see from the table, there are two sub-utilities available for use in this code: `addhexagons` and `addchevrons`.

Table 3.1: Initialization Variables

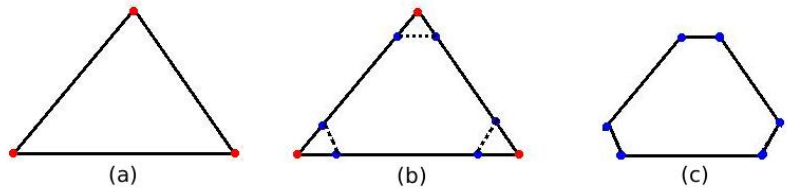
`handle`	'.g' Grid File Name
`n`	n^2 Gauss-Lobatto Nodes per Quadrilateral
`TOL`	Error Tolerance
`h`	Starting Step Size
`eps`	Epsilon for Approximating Derivative
`chevron`	Chevron Utility: 1 = On, 0 = Off
`hexagon`	Hexagon Utility: 1 = On, 0 = Off

The first sub-utility only affects corner polygons, which originally start as triangles. This utility changes these triangles into hexagons by splitting each singular point into two points and shifting the points equidistantly along the adjacent edges, as shown in Figure 3.1(a).

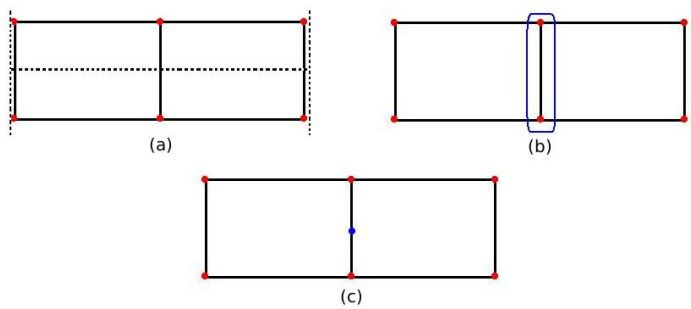
The second sub-utility works by isolating all polygons which exist partially in exactly two quadrilaterals from the original mesh. This happens only along quadrilateral edges of the original mesh. Once these polygons have been isolated, the utility then identifies the polygons which share the same edge - i.e., are next to each other. To this shared edge, the utility simply adds a point in the middle. This action, which turns the polygons into chevrons, allows an extra degree of freedom. See Figure 3.1(b). Because this utility requires there to be at least two neighboring polygons which share an edge, however, it must be turned off when `n = 3`.

In the first stage of iteration, we implement Euler's method to travel slightly in the direction of steepest descent for each point. For this stage, we use an adaptive step size h which increases each iteration that the l_2 error, calculated $l_2 = \sqrt{\frac{1}{2}\Sigma_i \frac{\varepsilon_i^2}{nuniq}}$ where ε_i is the error per i^{th} element and $nuniq$ is the number of unique elements, is less than the iteration before. If the l_2 error is larger than the previous iteration, h is decreased until l_2 once again declines. After the corners of the polygons are all moved, we must then normalize the new locations to make sure that they are still located upon the sphere. New surface areas are calculated and replace the old values. Then, using the new surface areas, we compute errors.

This repeats until the maximum absolute value of the errors dips below a hard-coded tolerance `NTOL` $= 10^{-5}$, at which point the second stage begins. In the second stage of iteration, in place of Euler's method, we use Newton's method to iteratively shift the corners to minimize the error. For this stage we define a new sparse matrix at every iteration and fill in 2nd order accurate derivatives of each point at specified locations. Then we use the built-in MATLAB function `x = lsqr(A,B,TOL,MAXIT)`, which attempts to solve the system of linear equations `Ax = B` where `A` is an mxn matrix, `B` is a column vector, `TOL` is the desired convergence tolerance, and `MAXIT` is the max number of allowed iterations. We



(a) AddHexagon: (i) Step 1 - Find triangles/corner polygons; (ii) Step 2 - Split each point into two distinct points; (iii) Step 3 - Move points equidistantly along adjacent edges



(b) AddChevrons: (i) Step 1 - Find polygons partially contained in exactly two original quadrilaterals; (ii) Step 2 - Isolate the shared edge between polygons; (iii) Step 3 - Insert a single point on the shared edge

Fig. 3.1: Sub-utility Process

then shift each point by calculating:

```
xyzcorners = xyzcorners - lsqr(dfdx, error, TOL, MAXIT)
```
(3.1)

where `xyzcorners` is the column vector of current corner locations ((x, y, z) Cartesian coordinates), `dfdx` is the sparse matrix filled with derivative values at each corner point, and `error` is the column vector of area error values for every polygon. The values `TOL` and `MAXIT` are hard-coded as the default MATLAB value 10^{-6} and 5000, respectively.

Euler's method runs first in order to reduce the error to a value low enough such that Newton's method can effectively converge the system in few iterations. We determined heuristically that the highest the error can be is 10^{-5}, as mentioned above, in order to work with all types of grids. For even the grids with the finest meshes, no more than 6 Newton's iterations were needed in order to achieve convergence. A check has been added to the code which will terminate the process should the number of Newton's iterations exceed 10. See Table 3.2 for a pseudocode of the algorithm.

3.2. Results. The results of this iterative method are extremely successful. Though the routine is relatively slow, taking roughly a day for the most refined grids attempted, the results exactly fit the criteria. We hoped that the method could reach a tolerance level `TOL` of 10^{-12}; we were glad to find, however, that it can actually achieve a `TOL` of 10^{-15}. Upon attempting any lower tolerance, the method stalls in the second stage of iteration and bounces back and forth between numbers barely over 10^{-16}.

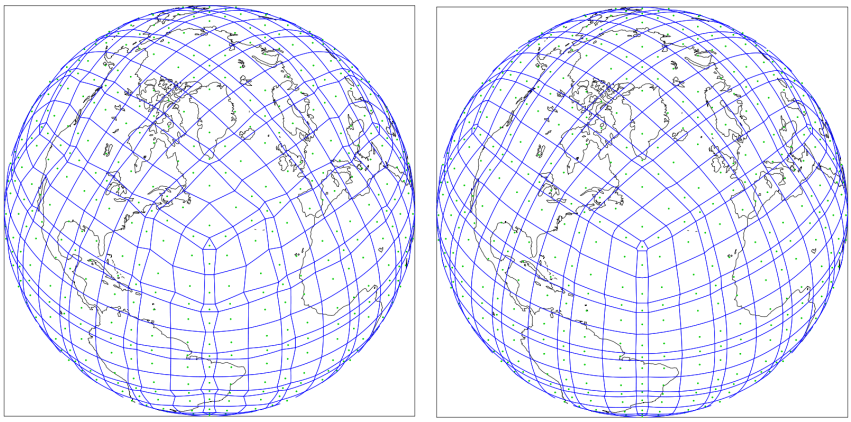
Our resulting dual grids for a 22.5° uniform mesh are displayed in Figures 3.2(a) and

Table 3.2: Iterative Steps

```
while max(abs(errort)) > TOL %Break statement
  initialize temporary areas
  if max(abs(errort)) > NTOL
  %Euler's Method using Steepest Descent
     initialize force matrix F
     for all unique Gauss-Lobatto Nodes
        calculate steepest descent F for corners
        and sum F for repeated corners
     end
     corners = corners + h*F
     normalize points back to surface of sphere
     calculate new surface area and new error
     calculate l2error
     if old l2error > new l2error
        h = 1.01*h
     else
        h = 0.3*h
     end
  else
  %Newton's Method using Least Squares
     initialize sparse matrix
     for all unique Gauss-Lobatto Nodes
        fill sparse matrix with
        2nd order accurate derivatives
     end
     corners = corners - lsqr(sparse matrix, error)
     normalize points back to surface of sphere
     calculate new surface area and new error
     if Newton Reps > 10
        error:  too many Newton iterations
     end
  end
  plot the resulting dual grid
end
```

3.2(b). Both grids were generated using the new mapping described in Section 2, utilizing the MATLAB mapping method from the section above. The grid in Figure 3.2(a) has the chevron and hexagon utilities deactivated (`chevron = hexagon = 0`), while the grid in Figure 3.2(b) has the sub-utilities activated (`chevron = hexagon = 1`).

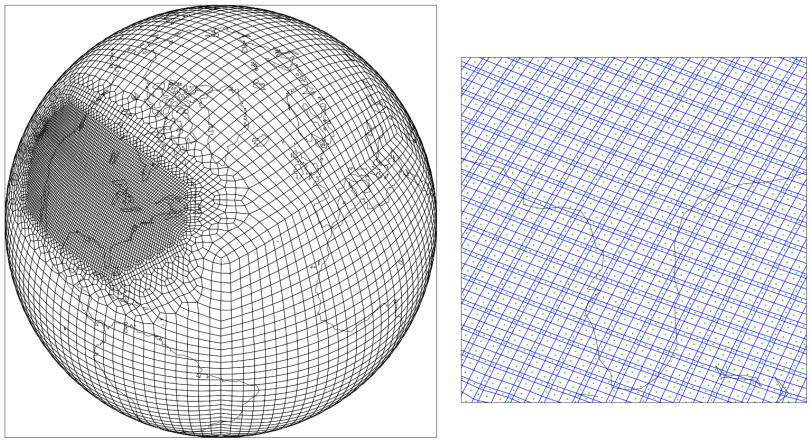
These grids also present the benefit of activating the sub-utilities. To create Figure 3.2(a), the run needed 6318 Euler iterations and 3 Newton iterations in order to converge to a tolerance level `TOL` $= 10^{-15}$, which took a real runtime of 2 hours 46 minutes. To create Figure 3.2(b), however, the run only needed 89 Euler iterations and 3 Newton iterations to converge to the same `TOL`, which took a real runtime total of 2 minutes 50 seconds, only 2% of the time without chevrons and hexagons. Scaled to a realistic mesh refinement (1° or finer), these utilities can save days or weeks of computation time.



(a) Chevron and Hexagon Utilities Off (b) Chevron and Hexagon Utilities On

Fig. 3.2: Comparison of Sub-Utility Effect on Dual Grids

The iterative method works for more than only uniform meshes, though. According to [9], the cost of running a simulation using a locally refined mesh as opposed to a globally refined mesh can be as much as 15 to 20 times less expensive computationally; therefore, it was vital that our method be able to create dual grids for these fundamentally essential mesh types. An example mesh can be seen in Figure 3.3(a). This grid, called CONUS, is a 1° globally refined mesh with a local refinement of $1/4^{\text{th}\circ}$ over the United States. Its dual grid is shown in Figure 3.3(b).



(a) CONUS: Original Mesh (b) Dual grid of CONUS, zoomed in over Florida

Fig. 3.3: CONUS Mesh and Resulting Dual Grid

4. Analysis.

4.1. Analytic Test Fields. Following the idealized study by [7], we begin our analysis with two test fields:

$$Y_{32}^{16} \quad = 2 + \sin^{16}(2\theta)\cos(16\lambda) \tag{4.1}$$

$$V_X = 1 - \tanh\left[\frac{\rho'}{d}\sin\left(\lambda' - \omega' t\right)\right]. \tag{4.2}$$

The first field (4.1) is a high frequency wave similar to a spherical harmonic of order 32 with azimuthal wavenumber 16. The second field (4.2) is a dual stationary vortex ([5]), where $\rho' = r_0 \cos\theta'$ with angular velocity

$$\omega'(\theta') = \begin{cases} 0 & \text{if } \rho' = 0 \\ \frac{V_t}{\rho'} & \text{if } \rho' \neq 0 \end{cases}$$

and normalized tangential velocity

$$V_t = \frac{3\sqrt{3}}{2}\text{sech}^2\rho' \tanh\rho'.$$

In these equations, (λ', θ') refers to a rotated spherical coordinate system with a pole located at (λ_0, θ_0). Following [7] and [3], we define $r_0 = 3$, $d = 5$, and $t = 6$. We differ, however, in that we define $(\lambda_0, \theta_0) = (0, 90)$ in order to rotate the vortex field over the fine local resolution of the CONUS mesh.

We have chosen these fields for one simple reason: the ability to compare our test data to the true values. We first calculate these fields on our chosen test grid, CONUS, seen in Figure 3.3(a). We then interpolate these fields using both ESMF algorithms and TempestRemap algorithms to a 1° latitude-longitude grid using four mapping methods: ESMF's conservative routine; ESMF's bilinear routine; TempestRemap's conservative method; and TempestRemap's conservative and monotone method. For the TempestRemap rountines, we utilize the `--in_meta` option, which allows the user to specify a metadata file – i.e., the dual grid created in Section 3.

We see from the highlighted rows in Tables 4.1 and 4.2 that while ESMF's conservative routine maintains the highest conservation between mappings, TempestRemap's conservative routine actually produces the lowest l_2 and l_∞ errors. Between the routines which are both conservative and monotone (ESMF conservative and TempestRemap conservative plus monotone), TempestRemap's algorithm yields superior results.

Table 4.1: Y_{32}^{16}: CONUS → 1° Lat-Lon

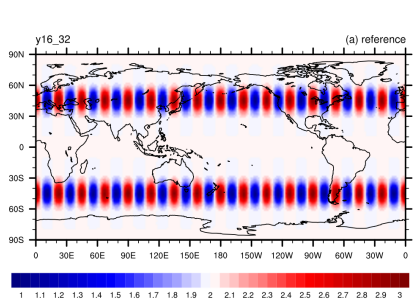
Map	**Rel. Cons. Err.**	**Rel. l_2 Err.**	**Rel. l_∞ Err.**
ESMF Cons	-8.8817841994e-16	0.0028276477	0.0350443549
ESMF Bilin	2.4314462348e-06	9.7528987551e-04	0.0677483902
Tempest Cons	6.0762506154e-13	2.5542887944e-04	0.0015392922
Tempest Cons&Mono	6.0762506154e-13	0.0020480475	0.0256390019

These results can also be verified in Figures 4.1(a) – 4.2(b). Figures 4.1(a) and 4.2(a) show the reference solutions on a slightly finer than $1/8^{th\circ}$ latitude-longitude mesh using ESMF's bilinear mapping method[6]. Figures 4.1(b) and 4.2(b) show the difference between the true values and the interpolated test data. As we can see for both figures, globally the most accurate method is TempestRemap's conservative routine.

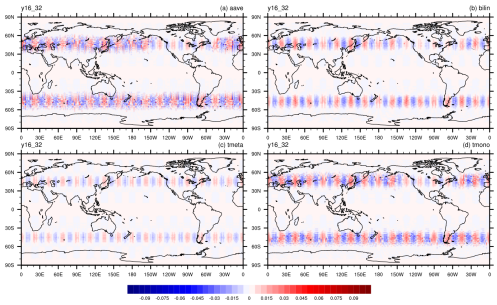
[6] Y_{32}^{16} and V_X were remapped to the finer latitude-longitude mesh in order to maintain consistency with results from the surface pressure and precipitation fields.

Table 4.2: V_X: CONUS → 1° Lat-Lon

Map	Rel. Cons. Err.	Rel. l_2 Err.	Rel. l_∞ Err.
ESMF Cons	-6.6613381450e-16	0.0026696437	0.0143002086
ESMF Bilin	-3.0253601009e-06	8.6118793680e-04	0.0087934835
Tempest Cons	6.1572968920e-13	3.6147702442e-04	0.0020496320
Tempest Cons&Mono	6.1550764460e-13	0.0017014020	0.0120947576



(a) Solution Plot



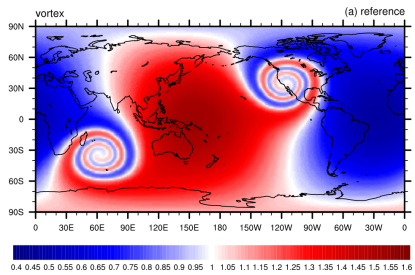
(b) Difference from True/reference solution: (i) ESMF Conservative routine; (ii) ESMF Bilinear routine; (iii) TempestRemap Conservative routine; (iv) TempestRemap Conservative and Monotone routine

Fig. 4.1: Y_{32}^{16} Plots

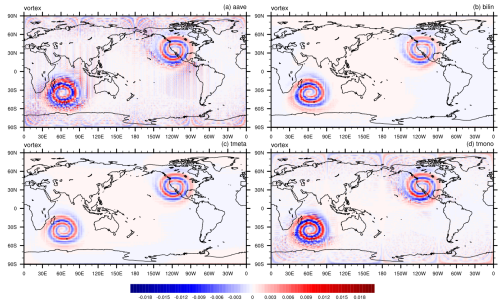
Looking specifically at Figure 4.2(b), though, we can see an example of an interesting phenomenon. One of the commonplace issues with ESMF's conservative routine is, when interpolating to a high resolution mesh, the resulting data takes on a "blocky" appearance. In Figure 4.2(b)(a), this is clearly apparent. Comparing to TempestRemap's version of conservative and monotone mapping (4.2(b)(d)), the blocky appearance is greatly reduced, instead producing a more "smeared" appearance like those of 4.2(b)(b), (c). This constitutes one of TempestRemap's advertised improvements in action.

4.2. Real Test Fields. After completing the analysis on the analytic test functions, we choose two real fields to analyze: surface pressure (PS) and precipitation (PRECT). These fields, unlike the test fields [(4.1), (4.2)], have no "true" values for a 1° latitude-longitude mesh against which we can compare. To handle this difficulty, we interpolate original CONUS data using ESMF's bilinear remapping routine to a slightly smaller than $1/8^{th}$ latitude-longitude grid. Since bilinear interpolation converges with a fine enough mesh, we consider this result to be the "true" data, which we call the reference data. We then follow a similar procedure to the analytic test fields. We interpolate PS and PRECT to a 1° latitude-longitude grid using the four mapping methods: ESMF's conservative routine; ESMF's bilinear routine; TempestRemap's conservative method, with `--in_meta` feature activated; and TempestRemap's conservative and monotone method, with `--in_meta` feature activated. In order to compare with the reference data, we then interpolate the four results to the same slightly smaller than $1/8^{th}$ latitude-longitude grid using ESMF's bilinear routine.

Error results for PS and PRECT can be seen in Tables 4.3 and 4.4, respectively. We



(a) Solution Plot



(b) Difference from True/reference solution: (i) ESMF Conservative routine; (ii) ESMF Bilinear routine; (iii) TempestRemap Conservative routine; (iv) TempestRemap Conservative and Monotone routine

Fig. 4.2: V_X Plots

see that, in both cases, ESMF's bilinear algorithm presents the lowest l_2 errors, though TempestRemap Conservative closely rivals these results, whereas TempestRemap routines yield the lowest l_∞ errors, highlighted in the tables. Again looking only at routines which are both conservative and monotone, TempestRemap produces the superior numbers.

Table 4.3: PS: CONUS → 1° Lat-Lon → 1\8° Lat-Lon

Map	**Relative l_2 Error**	**Relative l_∞ Error**
ESMF Conservative	0.002179606156281	0.054770216137821
ESMF Bilinear	0.001268259444092	0.039944410943513
Tempest Conservative	0.001273935767266	0.030386655114516
Tempest Cons. and Monotone	0.002004802651698	0.042339203632743

Table 4.4: PRECT: CONUS → 1° Lat-Lon → 1\8° Lat-Lon

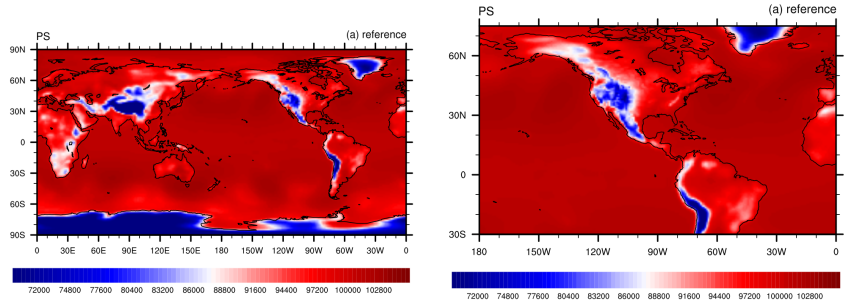
Map	**Relative l_2 Error**	**Relative l_∞ Error**
ESMF Conservative	0.138770011785557	0.352990491847132
ESMF Bilinear	0.110058601199370	0.353902362293653
Tempest Conservative	0.116629502370971	0.353456376203758
Tempest Cons. and Monotone	0.149044024540546	0.350990170744384

For these fields, we chose to create figures on a global and a local scale to mimic the levels of refinement of the CONUS mesh. For the PS field, Figures 4.3(a) and 4.4(a) show the original field and the difference from the reference solution, respectively, while Figures 4.3(b) and 4.4(b) show the localized field and difference from reference solution, respectively. We see that, much as expected, the least color variation occurs over the United States, where the CONUS mesh is more refined, a phenomenon that can also be easily seen in Figure 4.1(b) for Y_{32}^{16}.

For the PRECT field, parallel to the PS field, Figures 4.5(a) and 4.6(a) show the original

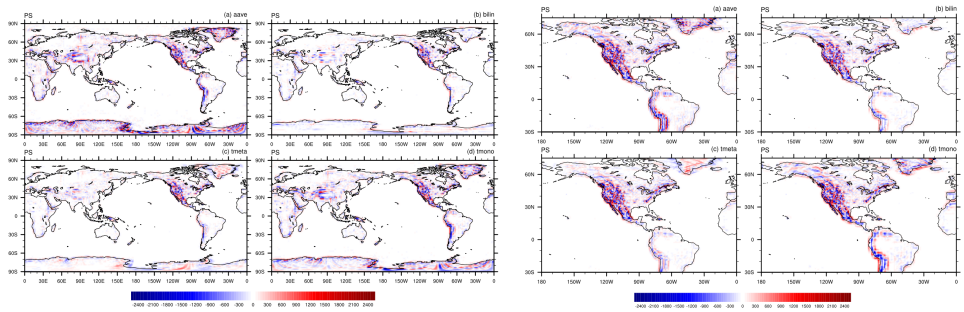
field and difference from reference solution, respectively. Figures 4.5(b) and 4.6(b) show the localized PRECT field and difference from the reference solution. The same smaller color variation phenomenon is present in these images, too, directly over the United States.

We find, as with the analytic test fields, that we can see the same "blocky" phenomenon in the ESMF conservative mapping routines (in Figures 4.4(a), 4.4(b), 4.6(a), and 4.6(b)). Again, we do not see any indication of the ESMF bilinear mapping issue (extraneous noise when remapping to low resolution).



(a) Global Solution (b) Solution over US

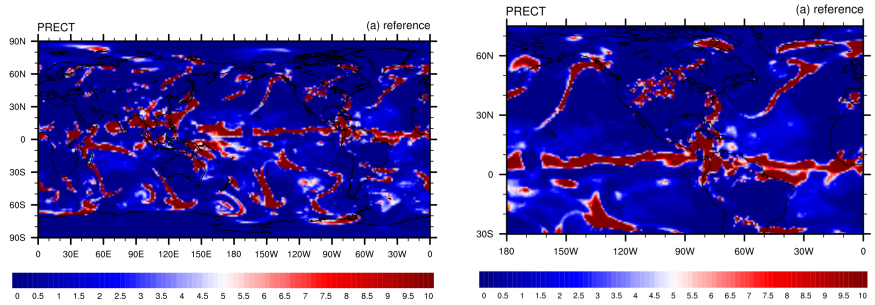
Fig. 4.3: Surface Pressure PS Solution Plots



(a) Difference from True/reference solution: (i) ESMF Conservative routine; (ii) ESMF Bilinear routine; (iii) TempestRemap Conservative routine; (iv) TempestRemap Conservative and Monotone routine

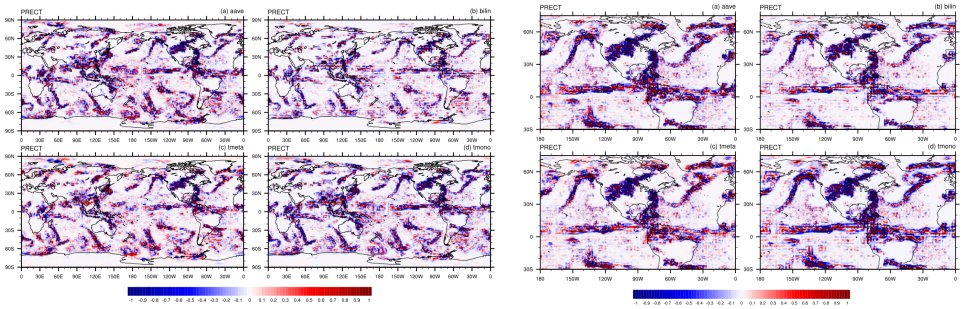
(b) Difference from True/reference solution: (i) ESMF Conservative routine; (ii) ESMF Bilinear routine; (iii) TempestRemap Conservative routine; (iv) TempestRemap Conservative and Monotone routine

Fig. 4.4: Surface Pressure PS Difference Plots



(a) Global Solution (b) Solution over US

Fig. 4.5: Precipitation *PRECT* Solution Plots



(a) Difference from True/reference solution: (i) ESMF Conservative routine; (ii) ESMF Bilinear routine; (iii) TempestRemap Conservative routine; (iv) TempestRemap Conservative and Monotone routine

(b) Difference from True/reference solution: (i) ESMF Conservative routine; (ii) ESMF Bilinear routine; (iii) TempestRemap Conservative routine; (iv) TempestRemap Conservative and Monotone routine

Fig. 4.6: Precipitation *PRECT* Difference Plots

5. Conclusion. Several conclusions arose during this study. First and foremost, we learned that adding extra degrees of freedom to the dual grid problem, by way of the sub-utilities `AddChevrons` and `AddHexagons`, we achieved a significant speedup in convergence. Because these dual grids are essential to HOMME, making them as accurate as possible is crucial - in addition to creating them in a reasonable amount of time. The proposed method has been shown to work for various test cases and produces good quality dual grids.

Our second conclusion is in reference to the tested remapping algorithms. We find that, for those algorithms which are required to be both conservative and monotone, TempestRemap bests ESMF. It both reduces error calculations and solves two well-recorded ESMF issues: blockiness when mapping to high resolution grids using the conservative algorithm and noisy data when mapping to low resolution using the bilinear algorithm. Though still in development, it presents exciting improvements to our remapping utilities.

REFERENCES

[1] O. Guba, M. A. Taylor, P. A. Ullrich, J. R. Overfelt, and M. N. Levy, *The spectral element method on variable resolution grids: Evaluating grid sensitivity and resolution-aware numerical viscosity*, Geoscience Model Development Discussion, Under Review (2014), pp. 1–27.

[2] P. W. Jones, *First- and second-order conservative remapping schemes for grids in spherical coordinates*, Monthly Weather Review, 127 (1999), pp. 2204–2210.

[3] P. H. Lauritzen and R. D. Nair, *Monotone and conservative cascade remapping between spherical grids (CaRS): Regular latitude-longitude and cubed-sphere grids*, Monthly Weather Review, 136 (2007), pp. 1416–1432.

[4] M. N. Levy, J. R. Overfelt, and M. A. Taylor, *A variable resolution spectral element dynamical core in community atmosphere model*, Tech. Rep. SAND: 2013–0697J, Sandia National Laboratories, 2013.

[5] R. D. Nair and B. Machenhauer, *The mass-conservative cell-integrated semi-lagrandian advection scheme on the sphere*, Monthly Weather Review, 130 (2002), pp. 649–667.

[6] C. Ronchi, R. Iacono, and P. S. Paolucci, *The 'cubed-sphere': A new method for the solution of partial differential equations in spherical geometry*, Journal of Computational Physics, 124 (1996), pp. 93–114.

[7] P. A. Ullrich and M. A. Taylor, *Arbitrary-order conservative and consistent remapping and a theory of linear maps, part 1*, 2014. In process.

[8] E. Williams, *Aviation formulary.* `http://williams.best.vwh.net/avform.htm`, 2011.

[9] C. M. Zarzycki, C. Jablonowski, and M. A. Taylor, *Using variable resolution meshes to model tropical cyclones in the community atmosphere model*, Monthly Weather Review, 142 (2014), pp. 1221–1239.

CROSS PLATFORM FINE GRAINED ILU AND ILDL FACTORIZATIONS USING KOKKOS

AFTAB Y. PATEL*, ERIK G. BOMAN†, SIVA RAJAMANICKAM ‡, AND EDMOND CHOW§

Abstract. In this paper we describe the implementation of a fine grained asynchronous algorithm for computing incomplete LU and LDL factorization preconditioners for sparse matrices. The algorithm is based on a reformulation of the factorization problem as the iterative solution of a nonlinear system of equations using an fixed point iterative method that has a high degree of inherent parallelism. The application of the factorization as a preconditioner is also achieved by using a basic iterative method. The approach exhibits significantly more parallelism than existing approaches and is particularly suited for many core architectures such as GPUs and the Intel Xeon Phi. The paper also describes various new techniques for improving the robustness of the algorithm thus enabling its effective application to real world problems.

1. Introduction. This paper is based on the fine grained parallel ILU algorithm developed in [6]. The algorithm is novel in the sense that it diverges significantly from the classical approach to computing incomplete LU factorizations. The algorithm in [6] and variations on it were implemented using the Kokkos framework [8]. The Kokkos framework is a set of libraries and tools designed to enable the development of cross-platform parallel software for many core architectures such as GPUs and CPUs, which traditionally have required the development of different implementations. The implementation of the fine grained ILU (known as FastILU from now on) using Kokkos was compiled for GPUs the Intel Xeon Phi and CPUs and tested against existing implementations on those platforms. Its performance was found to compare favorably to specialized code.

FastILU is an iterative method that constructs a sequence of approximations to the exact incomplete LU factorization using a simple fixed point iteration performed on the system of equations defining the ILU factorization. This reformulation results in a method which displays a high degree of parallelism since each non-zero of the new approximation in the sequence can be computed in parallel (in the limit of sufficiently many threads). While this results in a loss of accuracy, and consequently, preconditioner quality, due to the truncation of the iterations, an asynchronous update method such as those described in [9] can be used to alleviate most of these problems as described in [6].

The end objective of this work was the development of a preconditioning technique suitable for use in many core environments for the solution of real world problems. In its initial form as developed in [6] the algorithm had various shortcomings. The most significant was its failure on various problems arising from real world applications, due to numerical overflow. A number of different approaches were attempted to stabilize the fine grained algorithm without destroying its desirable properties. The first was the introduction of an under-relaxation parameter, which controlled the addition of a certain amount of previous values to the newly computed updates for the non-zeros of the incomplete factorization. The second was the use of a continuation method for computing the initial guesses for the iterative method with high levels of fill, using approximate solutions with lower levels of fill. Also the algorithm's convergence is sensitive to the assignment of computational work to threads. The use of a good assignment can yield good convergence. These techniques resulted in significant improvements in robustness. Most of the analysis in this paper will focus on these additions to the method and the demonstration of their effectiveness. To the

*School of Computational Science and Engineering, Georgia Institute of Technology, aypatel@gatech.edu
†Sandia National Laboratories, egboman@sandia.gov
‡Sandia National Laboratories, srajama@sandia.gov
§Advisor, School of Computational Science and Engineering, Georgia Institute of Technology, echow@cc.gatech.edu

best of our knowledge these contributions are new and represent a significant step forward in converting the experimental Fast-ILU algorithm into a tool ready for use by scientific computing researchers.

In addition to these improvements we developed an incomplete LDL method based on the original FastILU algorithm. This is particularly important in the context of GPU implementation because it avoids an expensive square root operation which is required in the IC method described in [6].

2. Background. The new parallel ILU algorithm is based on the sometimes-overlooked property that

$$(LU)_{ij} = a_{ij}, \quad (i,j) \in S \tag{2.1}$$

where $(LU)_{ij}$ denotes the (i,j) entry of the ILU factorization of the matrix with entries a_{ij}. In other words, the factorization is exact on the sparsity pattern S. The original ILU methods for finite-difference problems were interpreted this way [4, 14, 17] before they were recognized as a form of Gaussian elimination, and long before they were called incomplete factorizations [12].

Nowadays, an incomplete factorization is generally computed by a procedure analogous to Gaussian elimination. However, any procedure that produces a factorization with the above property is an incomplete factorization. The new fine-grained parallel algorithm interprets an ILU factorization as, instead of a Gaussian elimination process, a problem of computing unknowns l_{ij} and u_{ij} which are the entries of the ILU factorization, using property (2.1) as constraints.

Formally, the unknowns to be computed are

$$\begin{aligned} l_{ij}, &\quad i > j, \quad (i,j) \in S \\ u_{ij}, &\quad i \leq j, \quad (i,j) \in S. \end{aligned}$$

We use the normalization that L has a unit diagonal, and thus the diagonal entries of L do not need to be computed. Therefore, the total number of unknowns is $|S|$, the number of elements in the sparsity pattern S. To determine these $|S|$ unknowns, we use the constraints (2.1) which can be written as

$$\sum_{k=1}^{\min(i,j)} l_{ik} u_{kj} = a_{ij}, \quad (i,j) \in S. \tag{2.2}$$

Each constraint can be associated with an element of S, and therefore there are $|S|$ constraints. Thus we have a problem of solving for $|S|$ unknowns with $|S|$ equations.

To be sure, these equations are nonlinear and there are more equations than the number of rows in A. However, there are several potential advantages to computing an ILU factorization this way: 1) the equations can be solved in parallel with very fine-grained parallelism, 2) the equations do not need to be solved very accurately to produce a good ILU preconditioner, and 3) we often have a good initial guess for the solution.

We now discuss the parallel solution of the system of equations (2.2). Although these equations are nonlinear, we can write an explicit formula for each unknown in terms of the other unknowns. In particular, the equation corresponding to (i,j) can give an explicit formula for l_{ij} (if $i > j$) or u_{ij} (if $i \leq j$),

$$l_{ij} = \frac{1}{u_{jj}} \left(a_{ij} - \sum_{k=1}^{j-1} l_{ik} u_{kj} \right) \tag{2.3}$$

$$u_{ij} = a_{ij} - \sum_{k=1}^{i-1} l_{ik} u_{kj}. \tag{2.4}$$

The second of these equations does not need a divide by l_{ii} because $l_{ii} = 1$.

The above equations are in the form $x = G(x)$, where x is a vector containing the unknowns l_{ij} and u_{ij}. It is now natural to try to solve these equations via a fixed-point iteration

$$x^{(k+1)} = G(x^{(k)}) \tag{2.5}$$

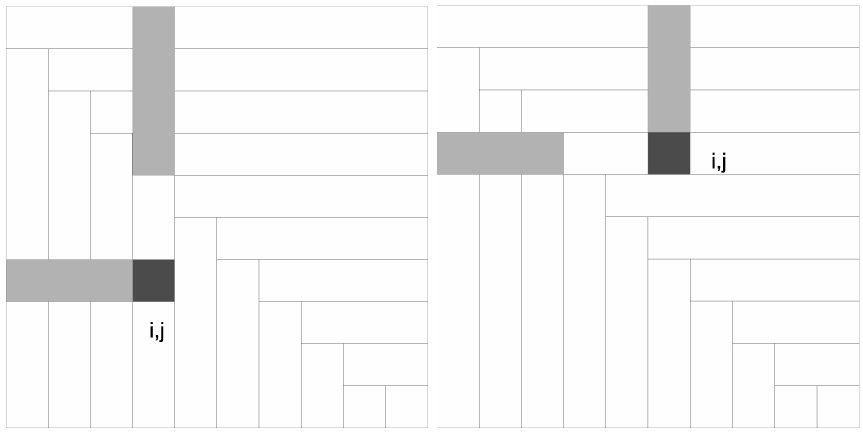
with an initial guess $x^{(0)}$. Each component of the new iterate $x^{(k+1)}$ can be computed in parallel.

There is a lot of structure in G. When the unknowns l_{ij} and u_{ij} are viewed as entries of matrices L and U, the formula (2.3) or (2.4) for unknown (i, j) only depends on other unknowns in row i of L to the left of j, and in column j of U above i. This is depicted in figure 2.1 where the L and U factors are shown superimposed into one matrix. Thus, an explicit procedure for solving the nonlinear equations *exactly* is to solve for the unknowns using equations (2.3) and (2.4) in a specific order: unknowns in the first row of U are solved (which depend on no other unknowns), followed by those in the first column of L; this is followed by unknowns in the second row of U and the second column of L, etc. This ordering could be called a "Gaussian elimination ordering," since it is one of the orderings in which the l_{ij} and u_{ij} are produced in Gaussian elimination. This ordering is just one of many topological orderings of the unknowns that could be used, another one being the natural "row-wise" ordering of the entries of A. To be clear, this ordering is not related to the reordering of the rows and columns of the matrix A, which we seek to avoid.

Different ways of performing the fixed-point iteration (2.5) in parallel gives rise to slightly different methods. If the components of $x^{(k+1)}$ are computed in parallel with only "old" values x, then the method corresponds to the nonlinear Jacobi method [15]. At the other extreme, if the components of $x^{(k+1)}$ are computed in sequence with the latest values of x, then we have the nonlinear Gauss-Seidel method. If this latter method visits the equations in Gaussian elimination order, then nonlinear Gauss-Seidel solves the equations in a single sweep, and the solution process corresponds exactly to performing a conventional ILU factorization. In practice, a parallel implementation may perform something in between these two extremes.

If the equations are ordered in a Gaussian elimination ordering the numerical method we have presented in this section converges in a finite number of iterations in exact arithmetic. A proof of this fact is presented in [6]. It is important to note at this juncture that this method may not converge due to numerical overflow in finite precision arithmetic, and the numerical experiments in later sections will demonstrate this. Our exposition in this section is necessarily brief, intended to give an overview of the method as the main focus of the present work is its cross-platform implementation and the improvement of its numerical stability, the reader should consult [6] for a more detailed development.

A question that remains is the application of the preconditioner that we have described in this section. The application of an ILU or LDL factorization in each step of an iterative method involves solve operations with the triangular factors. These solve operations present



(a) l_{ij} dependencies (b) u_{ij} dependencies

Fig. 2.1: Formula for unknown at (i, j) (dark square) depends on other unknowns left of (i, j) in L and above (i, j) in U (shaded regions). Left figure shows dependence for a lower triangular unknown; Right figure shows dependence for an upper triangular unknown.

some of the same challenges to parallelization that are encountered in the standard technique for constructing ILU factorizations. Thus while a parallel ILU algorithm is a nice thing to have, any benefit obtained from a highly parallel factorization operation would be nullified by poorly scalable triangular solves. In order to maintain a high degree of parallelism in the application of the factorization as a preconditiner we can solve with the triangular factors approximately using a fixed number of sweeps of a Jacobi iteration.

The Jacobi method is a basic iterative method based on the following matrix splitting

$$A = (A - D) + D; \tag{2.6}$$

where D is the diagonal of the matrix A. Given an initial guess x_0 Jacobi iteration generates a sequence of approximations $\{x\}_0^\infty$ to the solution of a linear system of equations $Ax = b$ according to the following update rule

$$x_{k+1} = (I - D^{-1}A)x_k + D^{-1}b. \tag{2.7}$$

The theory of basic iterative methods (of which class Jacobi iteration is a member) assures us that this iteration will converge if $\rho(I - D^{-1}A)) < 1$ [10]. $\rho(.)$ being the spectral radius of its argument. If A is triangular then $D^{-1}A$ is triangular and has a unit diagonal, implying that $(I - D^{-1}A)$ is a triangular matrix with a zero diagonal, which in turn implies that its spectral radius is zero. Thus, the Jacobi iteration converges for all triangular matrices and is a viable method for the application of an incomplete factorization with triangular factors. It is important to emphasize at this juncture that we are talking about *asymptotic* convergence here or convergence as the iteration number $k \to \infty$. Initially we may have divergence, or morbidly slow convergence depending on the matrix A and its eigenstructure.

3. Extensions and algorithms. It is easy to extend the numerical method outlined in the previous section to one for an incomplete LDL factorization for a symmetric matrix.

We start with the system of equations,

$$(LDL^T)_{ij} = a_{ij}, \quad (i,j) \in S. \tag{3.1}$$

In the above D is a diagonal matrix. We can use (3.1) to generate the fixed point iteration corresponding to the following equations for updating L and D,

$$l_{ij} = \frac{1}{d_j}\left(a_{ij} - \sum_{k=1}^{j-1} l_{ik} d_k l_{jk}\right) \tag{3.2}$$

$$d_i = a_{ii} - \sum_{k=1}^{i-1} l_{ik} d_k l_{jk}. \tag{3.3}$$

Note that d_i is the ith element on the diagonal of the diagonal matrix D. In contrast to the incomplete cholesky method developed in [6] this technique avoids square roots and has the potential to work for matrices that are indefinite.

The FastILU algorithm is presented in Algorithm 3 in pseudo-code. Each fixed-point iteration updating all the unknowns is called a "sweep."

Algorithm 3 Fine-Grained Parallel Incomplete Factorization

Set unknowns l_{ij} and u_{ij} to initial values
for sweep $= 1, 2, \ldots$ until convergence **do**
  for $(i,j) \in S$ **in parallel do**
    if $i > j$ **then**
      $l_{ij} = \left(a_{ij} - \sum_{k=1}^{j-1} l_{ik} u_{kj}\right) / u_{jj}$
    else
      $u_{ij} = a_{ij} - \sum_{k=1}^{i-1} l_{ik} u_{kj}$

The algorithm is parallelized across the elements of S. Given p compute threads, the set S is partitioned into p parts, one for each thread. The threads run in parallel, updating the components of the vector of unknowns, x, asynchronously. Thus the latest values of x are used in the updates. The work associated with each unknown is unequal but is known in advance (generally more work for larger i and j), and the load for each thread can be balanced.

In an implementation of the parallel for loop across S in algorithm 3 we have found experimentally that a distribution of iterations across threads where each thread iterates through a sequence of consequetive non-zeros in S or "chunk" generally yields the best convergence [6]. This would be expected to have a non-trivial impact on performance depending on the platform. In particular such a distribution of work between threads would be expected to perform poorly due to the lack of coalesced memory access. Our experimental results will demonstrate the impact of this.

To develop an efficient implementation of Algorithm 3, it is essential that sparsity is considered when computing (2.3) and (2.4). The inner products in these equations involve rows of L and columns of U. Thus L should be stored in row-major order (using compressed sparse row format, CSR) and U should be stored in column-major order (using compressed sparse column format, CSC). An inner product with a row of L and a column of U involves two sparse vectors, and this inner product should be computed utilizing their sparsity.

An additional optimization is to avoid the branch in Algorithm 3. This can be done by dividing the set S into upper and lower triangular parts; threads are then dedicated to either part.

When the matrix A is symmetric, the algorithm only needs to compute one of the triangular factors. The incomplete LDL^T algorithm that we described at the beginning of the section is presented in algorithm 4.

Algorithm 4 Symmetric Fine-Grained Parallel Incomplete Factorization

```
Set unknowns u_ij to initial values
for sweep = 1, 2, ... until convergence do
    for (i, j) ∈ S_U in parallel do
        s = a_ij − Σ_{k=1}^{j−1} l_ik d_k l_jk
        if i ≠ j then
            l_ij = s/d_j
        else
            d_i = s
```

In the above algorithms, we assume that the sparsity patterns S are given. Patterns corresponding to level-based ILU factorizations have proven to be effective for many types of problems. Patterns for ILU(k) for $k > 0$, however, must be computed sequentially; computing these patterns in parallel is an open problem. Many circumstances ameliorate the above problem. When solving a sequence of problems with the same sparsity pattern, the ILU pattern only needs to be computed once. Further, for problems on regular grids, the pattern for ILU(k) is the structure of the product of L and U for ILU($k-1$), and thus can be computed in parallel [16]. For irregular problems, this technique can be used as an approximation to the desired sparsity pattern.

Any fixed point iteration of the form (2.5) requires an initial guess to start. It is always one's hope that the algorithm be insensitive to the nature of this initial guess. In the present case unfortunately, this is not true. The iterative method, algorithm 3, is quite sensitive to the initial guess, as explained in [6]. One obvious initial guess is the upper and lower triangular parts of the matrix A. Experimentally it was observed that this was a poor choice. The problems with this choice of initial guess went away if the factorization method was applied to a scaled matrix $\tilde{A}$ obtained by scaling A to give it a unit diagonal (see [6]). While this is sufficient for constructing factorizations with low levels of fill, it yields to slow convergence to an effective factorization at higher levels of fill. A starting procedure that was found to be effective for high levels of fill, was to use the result of running the algorithm on a lower level of fill as a initial guess for the higher level factorizations. This was found to work well, as the experimental results will show.

At this point it is important to clarify the method of application of the preconditioner since we use a diagonally scaled matrix as an initial guess. Consider the scaled matrix $DAD = \tilde{A}$ and a corresponding incomplete factorization $\tilde{A} \approx LU$. Where L and U are approximate solutions to the system of equations

$$(\tilde{A} - LU)_S = 0 \tag{3.4}$$

where S is the given sparsity pattern. Now,

$$(\tilde{A} - LU)_S = 0 \tag{3.5}$$
$$\implies (DAD - LU)_S = 0 \tag{3.6}$$
$$\implies (AD - D^{-1}LU)_S = 0 \tag{3.7}$$
$$\implies (A - D^{-1}LUD^{-1})_S = 0. \tag{3.8}$$
$$\tag{3.9}$$

Note that the last two relationships in the above are only possible if D is a diagonal matrix. Thus the preconditioner can be applied from one side by applying $D^{-1}LUD^{-1}$.

4. Stabilization techniques. The main focus of the present work was making an implementation of FastILU that could be used by scientists to solve scientific computing problems arising from applications. One major pre-requisite of any method that is ready to be used in the real world is robustness. A user should ideally be able to use the method as a black box without worrying about its failure. This goal is, in most cases, unattainable however it is necessary to approximate it. In several places in the preceding sections there are comments about overflow and break-down of the FastILU algorithm. Now that we have completed an exposition of the algorithm itself we are ready to consider techniques that can be used to overcome some of its failings. The primary goal of this section will be to present several techniques that can be used to accomplish that goal without much effort. The techniques' effectiveness is explored in great detail in section 6.3.

A common technique used to stabilize the convergence of fixed point iteration is the concept of underrelaxation or damping. This involves replacing the iteration (2.5) by

$$x^{(k+1)} = (1-\omega)x^{(k)} + \omega G(x^{(k)}). \tag{4.1}$$

where $\omega \in (0,1]$. It can easily be seen that the iterations (2.5) and (4.1) have the same fixed points. Generally we expect that the convergence of the relaxed method will change depending on the choice of ω. Usually there is an optimal value of ω that depends on the function G that yields the best convergence. The jacobian of the right hand side of (4.1) controls the transient convergence or the change in the approximate solution $x^{(k)}$ from one iteration to the next. We can see immediately that this jacobian is nothing but ω times the jacobian of the right hand side of (2.5) plus $(1-\omega)$ times the appropriate identity. This has two main effects. Firstly since the jacobian of the right hand side of (2.5) has a spectral radius of 0 and is (strictly) lower triangular with a Gaussian Elimination ordering (see [6]) it sets the spectral radius to $1-\omega$, which for $\omega \in (0,1)$ is less than 1. Also, lower values of ω make the jacobian more diagonally dominant. The appropriate choice of ω could thus prevent the entries of the vector $x^{(k+1)}$ from being too large, which would in turn, prevent numerical overflow. Determining a value of ω a-priori is hard without specialized information about G (in the present context, this translates to information about A). The general trend that we observed is that ω can be chosen to make the method converge on most test cases. Generally as $\omega \to 0$ the rate of convergence slows down, making the cost of computing an incomplete factorization go up.

Another technique in the context of ILU that is used to construct factorizations for ill-conditioned factorizations is due to Manteuffel [11]. It involves scaling the off-diagonal entries of the matrix A using a parameter to make the scaled matrix $\tilde{A}$ more diagonally dominant. Subsequently the incomplete factorization constructed from the scaled matrix is used as a preconditioner for the original system. Once again this technique depends on a parameter which must be chosen depending on the matrix A, and it is expected that as the scaled and unscaled matrices depart significantly, the factorization of the scaled matrix will be less and less effective as a preconditioner for the unscaled matrix.

As was explained in [6] the convergence of the nonlinear iteration for computing the ILU factorization can be improved by controlling the assignment iterations of the parallel for loop in 3 to available threads. Making each thread process a set of these iterations (a chunk) in serial generally improves convergence if the non-zero indices in A are arranged in a Gaussian Elimination ordering. This leads to problems on GPU platforms however due to the lack of coalesced memory accesses when each thread processes a set of consecutive non-zeros. On CPUs the performance remains the same (or improves in some cases).

It was found that in certain cases the Jacobi iteration used for the triangular solve converged very slowly. One particular case is the matrix *bcsstk24* which is a symmetric and positive definite matrix with a very high condition number. In order to alleviate this problem we found that replacing the Jacobi iteration with a block-Jacobi iteration with small block sizes works quite well. Our experimental results will demonstrate this.

5. Kokkos implementation. Algorithms 3 and 4 were implemented using Kokkos. The algorithms in their present form are particularly suited for implementation using Kokkos. The bodies of the parallel for loops in the algorithms were implemented using functors, and the parallel for loops themselves were realized as using the *parallel_for* construct to deploy those functors.

For the Jacobi iteration used for the triangular solves the updates were implemented using functors that were also deployed using the *parallel_for* construct. The solution with the U factor produced by algorithm 3 was initially implemented to work with the CSC format in which the algorithm computes it. It was found that due to the necessity of the use of atomic operations this was considerably slower than the Jacobi iteration applied to L which is stored in CSR. Due to this we decided to compute and store the transpose of the U factor after its construction so that its application was efficient. It was clear from our timings that the benefits of this would far outweigh the cost of the transpose operation.

It is to be noted that no architecture-specific optimizations were added to our Kokkos implementation in order to test its default performance across architectures. It was found to perform as well as other specific implementations for GPUs and MIC architectures.

For testing the solution of the linear systems we also developed an interface for the new preconditioners to the Trilinos [2] preconditioning package called ifpack2. This was then used with the Conjugate Gradient and GMRES solver routines in the Trilinos solver package Belos.

6. Experimental results. We present the results of various experiments concerning the performance and preconditioning quality of our new preconditioning technique. It is to be noted that these results are generally a combination of the effect of the approximate incomplete factorizations and the approximate triangular solves used to apply them. Unless stated otherwise all the matrices were reordered using the RCM (Reverse Cuthill-McKee) ordering. The test matrices we used are given in tables 6.1 and 6.2. They are matrices arising from various application domains that are taken from the University of Florida Sparse Matrix collection [7].

Matrix	rows	non-zeros
thermal2	1228045	8580313
af_shell3	504855	17562051
ecology2	999999	4995991
apache2	715176	4817870
offshore	259789	4242673
G3_circuit	1585478	7660826
parabolic_fem	525825	3674625
bcsstk24	3562	159910

Table 6.1: Symmetric and positive definite matrices from the UFL sparse matrix collection.

6.1. Test devices.

1. GPU: Tesla K20Xm capability 3.5, Total Global Memory: 5.625 G, Shared Memory per Block: 48 K (Shannon).

Matrix	rows	non-zeros
chipcool0	20082	281150
venkat01	62424	1717792
atmosmodl	1489752	10319760
atmosmodd	1270432	8814880
FEM_3D_thermal 2	147900	3489300
stomach	213360	3021648

Table 6.2: Non-symmetric matrices from the UFL sparse matrix collection.

2. CPU: AMD Opteron 6276 2.3 Ghz 8x8 thread configuration (Vesper).
3. MIC: Intel Xeon Phi (KNC) with 228 threads (4 threads per core).

6.2. Timing experiments. In this section we present timing results for the FastILU method on different architectures. In order to provide a reference point we compared our implementation of the algorithm with two previous implementations. The first being the implementation in MAGMA [1] which is for GPUs. The details of this implementation can be found in [5] [3]. The second is an internal code developed at Georgia Tech (called GT-ILU), which was used for the experimental results in [6] and is for CPUs and Intel Xeon Phi (MIC). All the experiments here were carried out for the full ILU algorithm.

Table 6.3 is a comparison between the factorization construction time of the MAGMA implementation and our Kokkos implementation. t_{filu} are the times for the Kokkos implementation and t_{magma} are the times for MAGMA in seconds. The comparison was carried out by setting the number of sweeps to be used as 5. We can immediately see that the performance of the Kokkos implementation is comparable to the performance of the iterative ILU implementation in MAGMA.

Matrix	t_{magma} (s)	t_{filu} (s)	t_{filu}/t_{magma}
thermal2	0.045	0.045	1.00
af_shell3	0.401	0.405	1.01
ecology2	0.011	0.012	1.09
apache2	0.015	0.016	1.06
offshore	0.070	0.070	1.00
G3_circuit	0.021	0.022	1.05
parabolic_fem	0.019	0.019	1.00

Table 6.3: Timing comparisons of 5 sweeps of the full ILU between MAGMA and Kokkos implementation on a GPU. We see that the performance is similar.

Table 6.4 is a comparison between the factorization construction time of GT-ILU and our Kokkos implementation on the CPU device that we used for testing. We see immediately that our implementation is faster.

Table 6.5 is a comparison between the factorization construction time of GT-ILU and our Kokkos implementation on the MIC device that we used for testing. We see that the performance of both implementations is comparable.

Table 6.6 is a scaling study on one of the test problems from the matrices that we used for testing on a CPU. We see immediately that the Kokkos implementation scales significantly better than GT-ILU, despite being slower on a single core.

Tables 6.7 and 6.8 show the timings for triangular solves between our implementation

Matrix	t_{gtilu} (s)	t_{filu} (s)	t_{gtilu}/t_{filu}
thermal2	0.366	0.144	2.54
af_shell3	0.429	0.228	1.87
ecology2	0.130	0.061	2.13
apache2	0.133	0.064	2.07
offshore	0.233	0.080	2.91
G3_circuit	0.205	0.143	1.43
parabolic_fem	0.170	0.078	2.17

Table 6.4: Timing comparisons of 5 sweeps of the full ILU between old code and Kokkos implementation on a CPU. Note that different compilers were used for the two versions.

Matrix	t_{gtilu} (s)	t_{filu} (s)	t_{filu}/t_{gtilu}
thermal2	0.165	0.201	1.21
af_shell3	0.545	0.882	1.61
ecology2	0.079	0.068	0.86
apache2	0.088	0.078	0.89
offshore	0.113	0.124	1.10
G3_circuit	0.126	0.131	1.04
parabolic_fem	0.083	0.070	0.84

Table 6.5: Timing comparisons of 5 sweeps of the full ILU between old code and Kokkos implementation on MIC with 228 threads.

and MAGMA on a GPU and our implementation on MIC. On a GPU the timings are comparable. Comparing the times of the CSR solve (for L) vs the CSC solve (for U) we immediately see that the CSR solve is up-to a factor of 4 faster on some cases due to the lack of atomics. This is the main motivation for us to compute the transpose of U (which is equivalent to converting it to CSR) after computing L and U. If we expect that U is going to be applied many times in the course of the solution of a linear system, then the profits from using a faster triangular solve far outweigh the cost of the transpose. For the MAGMA GPU implementation we also timed the transpose operation which is a call to a CUSPARSE library function [13] (the timings are given in the column with header t^T_{magma}). We did not include the timings for the tranpose in our FastILU code because we did not optimize this operation.

In section 4 we claimed that increasing the chunk size for the nonlinear iterations to construct the ILU factorization resulted in improved convergence but led to degraded performance on GPU platforms. Figure 6.1 illustrates the performance impact of increasing the chunk size on GPUs on the problem *af_shell3*. We see that the lack of coalesced memory accesses results in a significant performance degradation.

6.3. Convergence and preconditioning performance. In this section we present various experimental results to demonstrate the convergence and preconditioning quality of the new algorithm. These experiments will motivate our choice of stabilization methods and the new initial guess (described in sections 3 4). The focus of our presentation in this section will be a GPU platform since the convergence of the method degrades with increasing number of threads and our GPU platform had a greater number of threads than the MIC and CPU platforms (see [6] for a fairly detailed analysis of this fact). When we refer to the preconditioner being used with a certain number of sweeps we are referring to the number

threads	t_{gtilu} (s)	t_{filu} (s)
1	1.745	3.089
8	0.423	0.531
16	0.391	0.361
32	0.377	0.201
40	0.374	0.170
48	0.368	0.180
56	0.370	0.163
64	0.362	0.153

Table 6.6: Timing comparisons of 5 sweeps of the full ILU between old code and Kokkos implementation on a CPU with varying numbers of cores. The matrix used is *thermal2*

Matrix	t_{magma} (s)	t^T_{magma}	t^L_{filu} (s)	t^U_{filu} (s)
thermal2	0.0068	0.1011	0.0079	0.0249
af_shell3	0.0172	0.1640	0.0193	0.0711
ecology2	0.0028	0.0588	0.0038	0.0056
apache2	0.0029	0.0560	0.0034	0.0067
offshore	0.0043	0.0412	0.0048	0.0169
G3_circuit	0.0045	0.0909	0.0061	0.0118
parabolic_fem	0.0026	0.0422	0.0031	0.0081

Table 6.7: Comparisons of timings of triangular solves between MAGMA and the Kokkos implementation on a GPU. Note that MAGMA transposes the matrix U after construction so that all solves are CSR solves whereas the Kokkos implementation uses a CSR solve and a CSC solve. The timings shown are for 5 iterations of Jacobi's method applied to the triangular factors of a level 0 factorization. Timings were averaged over 100 repetitions. t^T_{magma} is the time that MAGMA takes for computing the transpose of the U factor.

of sweeps being used for the factorization phase and the number of Jacobi iterations used for application. These two numbers are assumed to be the same in the remainder of this section. Also, we would like to point out that when we refer to the *nonlinear residual* we are referring to the ILU residual $\|(A - LU)_S\|_F$. Here $(A)_S$ is the pattern of A restricted to the sparsity pattern S (i.e. the elements of A not in S are dropped).

Tables 6.9 and 6.10 present the number of CG iterations for the solution of various test problems with the LDL preconditioner for various levels of fill. These results show that for certain problems *offshore* and *af_shell3* the preconditioning technique fails. The failures in these cases were observed to be due to a breakdown of the non-linear asynchronous iterations for constructing the factorization due to numerical overflow.

In an attempt to stabilise the method and solve the problems with breakdown we used under-relaxation with a parameter ω value of 0.5. This solved the problems with the breakdowns observed with the unrelaxed method. These results are presented in tables 6.11 and 6.10 for 5 and 10 sweeps respectively. The variation of the l2 norm of the ILU residual with the relaxation parameter and the variation of the number of CG (conjugate gradient) iterations for *offshore* and *af_shell3* is presented in figures 6.2(a), 6.2(b), 6.3(a) and 6.3(b). These figures highlight how ω affects the preconditioning technique. Lower values of omega generally lead to slower, more stable convergence to an ILU factorization. In all cases the preconditioned CG convergence tolerance used was 10^{-6}.

Tables 6.11 and 6.12 reveal another problem that is present in tables 6.9 and 6.10 as

Matrix	t^L_{filu} (s)	t^U_{filu} (s)
thermal2	0.0292	0.0903
af_shell3	0.0128	0.0719
ecology2	0.0168	0.0303
apache2	0.0139	0.0286
offshore	0.0088	0.0367
G3_circuit	0.0289	0.0587
parabolic_fem	0.0121	0.0426

Table 6.8: Timings of the triangular solves for the Kokkos Implementation on MIC with 228 threads. Timings were averaged over 100 repetitions.

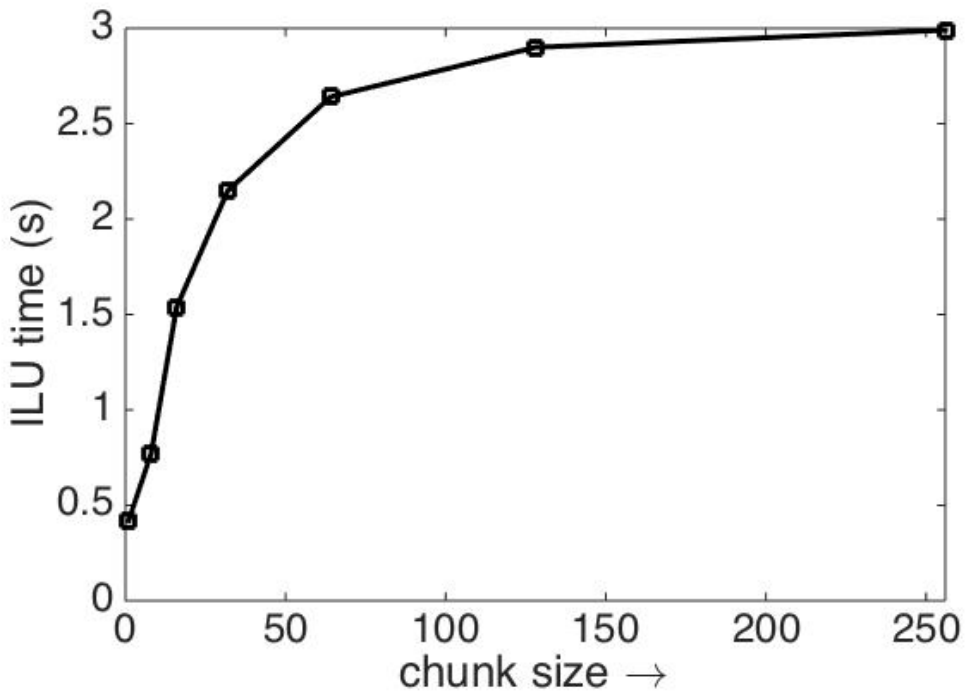


Fig. 6.1: Effect of increasing chunk size on factorization time on the GPU with 5 sweeps of the asynchronous iterations. Test problem used is *af_shell*

well. The increasing levels of fill do not give us better pre-conditioners if the number of sweeps used is kept constant. This becomes very evident when we compare these results to the results from using an exact ILU factorization, presented in table 6.17. In an attempt to resolve this problem we implemented a continuation type technique for determining the initial guess for higher levels of fill based on the factorization constructed using the technique at lower levels of fill. The guess for a level k factorization was taken as the result of applying three sweeps of the FastILU algorithm with level $k-1$. This is recursively continued until level 0. The effect of using such a procedure for constructing initial guesses on the matrix *apache2* is presented in tables 6.13 and 6.14. We see that the effect of using the new initial

Matrix	0	1	2	3	4	5
thermal2	1421	1110	1086	1145	1172	1178
af_shell3	*	*	*	*	*	*
ecology2	1807	1311	1271	1300	1344	1308
apache2	1001	768	815	818	827	847
offshore	*	*	*	*	*	*
G3_circuit	868	612	586	574	568	562
parabolic_fem	425	467	421	474	480	527

Table 6.9: Test of the Iterative LDL preconditioner with increasing factorization level. The configuration of the preconditioner is 5 sweeps for both construction and triangular solve. All matrices have been re-ordered using the RCM ordering. No under-relaxation is used.

Matrix	0	1	2	3	4	5
thermal2	1312	869	692	635	630	626
af_shell3	*	*	*	*	*	*
ecology2	1708	1082	832	761	738	723
apache2	967	541	326	299	293	294
offshore	334	*	*	*	*	*
G3_circuit	860	524	426	360	312	254
parabolic_fem	334	271	255	249	263	292

Table 6.10: Test of the Iterative LDL preconditioner with increasing factorization level. The configuration of the preconditioner is 10 sweeps for both construction and triangular solve. All matrices have been re-ordered using the RCM ordering. No under-relaxation is used.

guess procedure, compared to the initial technique using the upper and lower triangular parts of the scaled matrix, is significant on this particular case. Tables 6.15 and 6.16 show that this improvement is observed on all our test problems and is not specialized to the case of *apache2*. Comparing tables 6.14 and 6.17 we observe another interesting fact. The last row of 6.14 has uniformly lower iteration counts than the exact case. Indeed there is nothing in the theory to prevent this from happening, and this suggests that the inexact LDL method may, in come cases be a better preconditioner than an exact ILU factorization.

We now consider the effect of the Manteuffel shifting that we described in section 4. The shifting parameter was varied in increments of 0.01 for the matrices *af_shell3* and *offshore* which have convergence issues on GPUs. In order to isolate the effects of shifting we did not use underrelaxation on these tests. The variation of the number of CG iterations with the shifting parameter for these problems is presented in figures 6.4(a) and 6.4(b). We see that fairly large shifting parameters ranging from 0.08 to 0.20 can be used without much decrease in iterations resulting from the use of the incomplete factorization of the shifted matrix as a preconditioner for the original system. The algorithm used to compute the factorization used for the results in figures 6.4(a) and 6.4(b) was the LDL algorithm.

Next we consider a particularly ill-conditioned symmetric and positive definite matrix *bcsstk24* with a condition number $\kappa(A) \sim 10^{11}$. While the ILU factorization is found to converge the jacobi iteration for the triangular solves converges very slowly.In order to allevaite this problem we used a block jacobi triangular solve. Reasonably small block sizes yield good results. Table 6.18 shows the effect of increasing the block size used for the triangular solves from 5 to 20. The factorization used is a ILU level 1 factorization since the ILU(0) factorization is a very poor preconditioner. The exact level 1 ILU factorization with

Matrix	0	1	2	3	4	5
thermal2	1489	1174	1156	1171	1192	1199
af_shell3	1128	991	846	940	825	861
ecology2	1844	1444	1419	1440	1402	1464
apache2	1267	814	757	821	786	867
offshore	385	301	304	290	291	290
G3_circuit	958	702	644	635	624	615
parabolic_fem	461	489	453	440	453	442

Table 6.11: Test of the Iterative LDL preconditioner with increasing factorization level. The configuration of the preconditioner is 5 sweeps for both construction and triangular solve. All matrices have been re-ordered using the RCM ordering. The under-relaxation parameter used is $\omega = 0.5$.

Matrix	0	1	2	3	4	5
thermal2	1343	924	840	815	819	811
af_shell3	901	653	565	589	554	599
ecology2	1704	1103	925	910	893	922
apache2	1043	629	432	484	427	497
offshore	350	211	184	175	172	172
G3_circuit	904	607	512	471	431	410
parabolic_fem	356	328	295	288	285	286

Table 6.12: Test of the Iterative LDL preconditioner with increasing factorization level. The configuration of the preconditioner is 10 sweeps for both construction and triangular solve. All matrices have been re-ordered using the RCM ordering. The under-relaxation parameter used is $\omega = 0.5$.

Sweeps	0	1	2	3	4	5
5	1271	818	758	821	786	865
10	1043	630	430	486	426	497
15	984	557	361	341	301	324
20	967	537	330	310	271	277
25	967	533	311	295	253	253

Table 6.13: Number of solver iterations with different numbers of sweeps and levels for *apache2* example re-ordered using RCM. The standard initial guess was used. The under-relaxation parameter used was $\omega = 0.5$.

Sweeps	0	1	2	3	4	5
5	1268	755	539	547	624	722
10	1043	592	370	290	267	267
15	984	545	334	250	203	191
20	967	536	306	235	188	168
25	967	533	317	230	181	159

Table 6.14: Number of solver iterations with different numbers of sweeps and levels for *apache2* example re-ordered using RCM. The continuation method for initial guesses was used. The under-relaxation parameter used was $\omega = 0.5$.

Matrix	0	1	2	3	4	5
thermal2	1489	1107	1047	1073	1088	1095
af_shell3	1134	1012	975	969	953	966
ecology2	1841	1416	1282	1269	1303	1306
apache2	1265	756	539	547	625	721
offshore	384	288	279	308	373	364
G3_circuit	957	678	573	542	531	532
parabolic_fem	467	436	405	429	455	474

Table 6.15: Test of the Iterative LDL preconditioner with increasing factorization level. The configuration of the preconditioner is 5 sweeps for both construction and triangular solve. All matrices have been re-ordered using the RCM ordering. The continuation guess is used along with under-relaxation with $\omega = 0.5$.

Matrix	0	1	2	3	4	5
thermal2	1343	904	727	646	623	608
af_shell3	902	666	569	534	547	424
ecology2	1704	1114	879	799	759	725
apache2	1043	592	370	291	267	267
offshore	350	205	178	178	282	274
g3_circuit	903	567	498	419	357	291
parabolic_fem	358	296	253	246	251	256

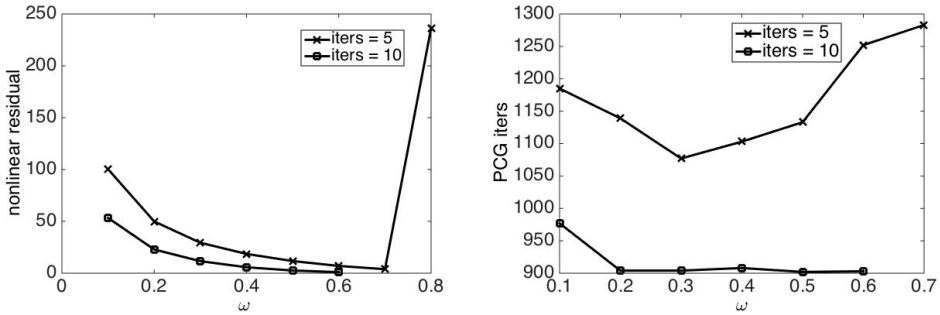
Table 6.16: Test of the iterative LDL preconditioner with increasing factorization level. The configuration of the preconditioner is 10 sweeps for both construction and triangular solve. All matrices have been re-ordered using the rcm ordering. The continuation guess is used along with under-relaxation with $\omega = 0.5$.

matrix	0	1	2	3	4	5
thermal2	1934	1225	856	637	507	440
af_shell3	1248	788	583	462	369	309
ecology2	1625	988	696	576	467	414
apache2	1294	619	394	289	235	188
offshore	485	*	*	*	*	*
g3_circuit	1414	757	546	421	341	303
parabolic_fem	313	238	164	129	106	91

Table 6.17: Test of the exact ilu preconditioner with increasing factorization level. All matrices have been re-ordered using the RCM ordering

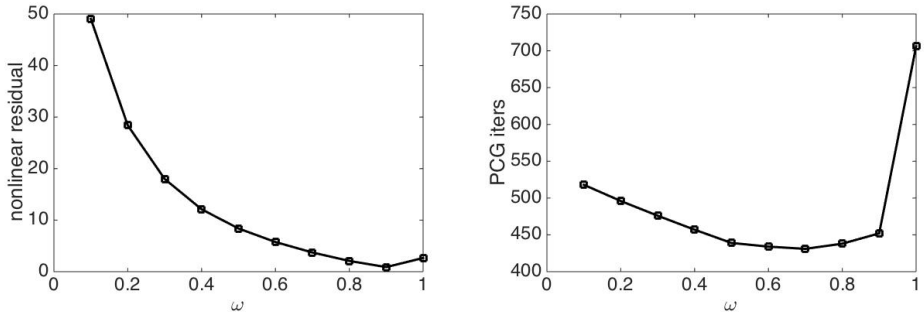
blk size	5	10	15
5	*	*	*
10	*	258	147
15	*	153	105
20	479	102	94

Table 6.18: Effect of the block size used in the block Jacobi triangular solves on solver iterations for the matrix *bcsstk24*



(a) Nonlinear residual vs. ω (b) Iterations vs. ω

Fig. 6.2: Effect of underrelaxation parameter ω on nonlinear residual and solver iterations for *af_shell3*



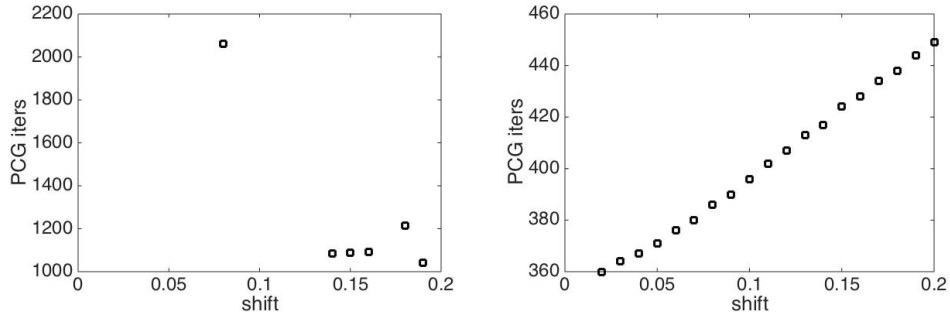
(a) Nonlinear residual vs. ω (b) Iterations vs. ω

Fig. 6.3: Effect of underrelaxation parameter ω on nonlinear residual and solver iterations for *offshore*

exact triangular solves yields a iteration count of 93 iterations. The block Jacobi method was parallelized in the same way as the Jacobi solve.

6.4. Non-symmetric test cases. Upto now the experimental results have focussed mainly on matrices that are symmetric and positive definite. This section will present results of using the new factorization method on the non-symmetric matrices listed in 6.2. In all cases the solver used was GMRES restarted after 50 iterations with a convergence tolerance of 10^{-6}. These results are summarized in table 6.19. The solver used was the one implemented in the Belos package of Trilinos. The corresponding results with an under-relaxation parameter $\omega = 0.5$ are shown in table 6.20. These tests were carried out on our GPU test platform, and for these cases the method of application of the preconditioner was block-Jacobi with a block size of 20.

We see that the method breaks down for two of the six non-symmetric test cases. In the remaining cases however its behaviour seems to conform to what we observed for the LDL method.



(a) *af_shell3* (b) *offshore*

Fig. 6.4: Variation of solver iterations with shift for *af_shell3* and *offshore*. 5 sweeps were used for the factorization and triangular solves.

matrix	lvl 0 (5)	lvl 0 (10)	lvl 1 (5)	lvl 1 (10)	lvl 0 ex	lvl 1 ex
atmosmodl	80	76	61	51	73	48
atmosmodd	195	189	153	100	186	90
stomach	*	*	*	*	10	5
venkat01	*	*	*	*	15	11
FEM_3D_thermal 2	10	8	9	6	8	6
chipcool0	86	75	43	39	75	39

Table 6.19: Solver iterations for non-symmetric test cases. The tests are for level 0 and level 1 matrices. The numbers in brackets indicate the number of sweeps used for the iterative ILU method without under-relaxation and with a block size of 4096. "ex" denotes the exact ILU preconditioner. The solver used was GMRES restarted at 50 iterations with a convergence tolerance of 10^{-6}.

matrix	lvl 0 (5)	lvl 0 (10)	lvl 1 (5)	lvl 1 (10)	lvl 0 ex	lvl 1 ex
atmosmodl	88	76	76	53	73	48
atmosmodd	236	190	185	104	186	90
stomach	*	*	*	*	10	5
venkat01	*	*	*	*	15	11
FEM_3D_thermal 2	10	8	8	6	8	6
chipcool0	98	81	78	42	75	39

Table 6.20: Solver iterations for non-symmetric test cases. The tests are for level 0 and level 1 matrices. The numbers in brackets indicate the number of sweeps used for the iterative ILU method with $\omega = 0.5$ and with a block size of 4096. "ex" denotes the exact ILU preconditioner. The solver used was GMRES restarted at 50 iterations with a convergence tolerance of 10^{-6}.

7. Conclusions and future work. In this report we presented the implementation of a fine grained parallel asynchronous technique for the construction of incomplete ILU and LDL factorizations. We also presented a parallel technique for applying the preconditioner in a parallel environment. Except for the LDL extension the technique in this papaer was previously developed in [6]. The authors of that paper, however, did not use GPUs for testing. Running the algorithm on GPUs exposed various robustness problems that were mainly due to the particular thread assignment required for maximum efficiency, and the large number of available threads. The technique initially implemented was not numerically very robust and broke down on several of our test cases. We utilized under-relaxation, and a continuation procedure for constructing initial guesses in order to alleviate these problems. These measures were found to be effective and served to greatly increase the robustness of the method.

For our implementation we used the Kokkos framework. It was found that the Kokkos framework offers one flexibility similar to CUDA and OpenMP. In the present case the implementation of the algorithm was not constrained by our choice of Kokkos as our implementation platform. This work makes a convincing case for the adoption of Kokkos for cross platform programming targeted at many core architectures.

There are several avenues of future work that remain open. The first is the development of a block version of the iterative method and the second is the investigation of the effectiveness of optimization based approaches to constructing approximate factorizations such as stochastic gradient descent, in the context of ILU factorizations. The theoretical understanding of the iterative ILU method is also severely lacking. There seems to be no clear theoretical result that explains the remarkably good performance of the method even when the nonlinear residual is large. The development of such theory would also constitute a possible future topic of research.

8. Acknowledgements. The authors would like to acknowledge the reviewer, A. M. Bradley, for his helpful comments and suggestions that led to non-trivial improvements in this report.

REFERENCES

[1] *MAGMA*. `http://icl.cs.utk.edu/magma/index.html`, 2015. [Online; accessed 19-July-2015].
[2] *Trilinos*. `http://www.trilinos.org`, 2015. [Online; accessed 19-July-2015].
[3] H. Anzt, E. Chow, and J. Dongarra, *Iterative sparse triangular solves for preconditioning*, (to appear).
[4] N. Buleev, *A numerical method for the solution of two-dimensional and three-dimensional equations of diffusion*, Math. Sb, 51 (1960), p. 15.
[5] E. Chow, H. Anzt, and J. Dongarra, *Asynchronous iterative algorithm for computing incomplete factorizations on gpus*, in Lecture Notes in Computer Science (LNCS), vol. 9137, Springer, 2015, pp. 1–16.
[6] E. Chow and A. Patel, *Fine-grained parallel incomplete lu factorization*, SIAM Journal on Scientific Computing, 37 (2015), pp. C169–C193.
[7] T. A. Davis and Y. Hu, *The university of florida sparse matrix collection*, ACM Transactions on Mathematical Software (TOMS), 38 (2011), p. 1.
[8] H. C. Edwards, C. R. Trott, and D. Sunderland, *Kokkos: Enabling manycore performance portability through polymorphic memory access patterns*, Journal of Parallel and Distributed Computing, 74 (2014), pp. 3202–3216.
[9] A. Frommer and D. B. Szyld, *On asynchronous iterations*, Journal of computational and applied mathematics, 123 (2000), pp. 201–216.
[10] C. T. Kelley, *Iterative methods for optimization*, vol. 18, Siam, 1999.
[11] T. A. Manteuffel, *An incomplete factorization technique for positive definite linear systems*, Mathematics of computation, 34 (1980), pp. 473–497.

[12] J. A. Meijerink and H. A. van der Vorst, *An iterative solution method for linear systems of which the coefficient matrix is a symmetric m-matrix*, Mathematics of computation, 31 (1977), pp. 148–162.

[13] M. Naumov, L. Chien, P. Vandermersch, and U. Kapasi, *Cusparse library*, in GPU Technology Conference, 2010.

[14] T. A. Oliphant, *An implicit, numerical method for solving two-dimensional time-dependent diffusion problems*, Quart. Appl. Math., 19 (1961).

[15] J. Ortega and W. Rheinboldt, *Iterative Solution of Nonlinear Equations in Several Variables*, Academic Press, New York, 1970.

[16] Y. Saad, *Iterative Methods for Sparse Linear Systems*, SIAM, Philadelphia, PA, 2nd ed., 2003.

[17] R. S. Varga, *Factorization and normalized iterative methods*, tech. rep., Westinghouse Electric Corp. Bettis Plant, Pittsburgh, 1959.

Applications

Articles in this section discuss the use of computational techniques such as those discussed in the previous section to simulate physical systems.

Bynum, Klise, Laird, Murray, Seth, and Siirola describe their new Water Network Tool for Resilience (WNTR). It is intended to help evaluate and improve the resilience of water networks, a critical infrastructure. The focus of this report is the WNTR's hydraulic model. WNTR is implemented in Python and uses the modeling language Pyomo.

Go, Muñoz, and Watson assess the economic value of grid-scale energy storage systems as part of future generation, storage, and transmission systems. They formulate their stochastic model as a Mixed Integer Linear Problem (MILP).

Newton, Rintoul, Valicka, and Wilson describe a new destination prediction algorithm and apply it to predicting an aircraft's destination based on the first half of its trajectory. The algorithm builds a high-dimensional feature space using historical data and then uses the method of nearest neighbors to make predictions.

Porter and Mousseau perform an uncertainty quantification study of the interfacial mass transfer model in the light water reactor simulator CTF, a legacy code. They describe an example of holistic uncertainty analysis using Bayesian calibration. The example is the expression used to compute the vapor heat transfer coefficient for large bubbles and droplets and for superheated small bubbles. The expression contains four uncertain parameters; the authors use a number of experiments to estimate distributions for these.

Roberts, Mitchell, Thompson, and Tikare add two new capabilities to the Stochastic Parallel Particle Kinetic Simulator (SPPARKS), a kinetic Montel Carlo material simulator: the grain curvature diagnostic, and a temperature gradient extension of the Potts model for simulating grain growth in metals.

Wang and Rothganger consider graph-theoretic aspects of the representation of state and structural dynamics in spike timing-based neural networks. One application of these ideas is to software simulations of neural networks.

A.M. Bradley
M.L. Parks

December 18, 2015

WATER NETWORK HYDRAULICS WITH PRESSURE-DEPENDENT DEMAND FOR WNTR: A WATER NETWORK TOOL FOR RESILIENCE

MICHAEL L. BYNUM*, KATHERINE A. KLISE†, CARL D. LAIRD‡, REGAN MURRAY§, ARPAN SETH¶, AND JOHN D. SIIROLA‖

Abstract. Water networks are critical infrastructure. As such, damage to and/or contamination of water networks can cause harm to entire communities. Tools for evaluating and improving the resilience of water networks to adverse events are vital to protecting inhabitants of the United States. We are developing just such a tool, WNTR: Water Network Tool for Resilience. In this paper, we review the models used in WNTR to simulate the water network hydraulics, including pressure-dependent demand and pipe leaks.

1. Introduction. Communities around the world depend heavily on water distribution systems to be reliable. Damage to the network (broken pipes, power outages, etc.) can create a deficiency in the water supply. The U.S. EPA [4] defines the resilience of water networks as "the ability of the human organizations that manage water to design, maintain, and operate water infrastructure (e.g., water sources, treatment plants, storage tanks, and distribution systems) in such a way that limits the effects of disasters on the water infrastructure and the community it serves, and enables rapid return to normal delivery of safe water to customers." Water utilities need software tools to evaluate network performance during and after various disaster scenarios to assist in making operational decisions to improve resilience (e.g., decide which leaks to repair first). Demand driven models, such as EPANET, are not adequate to simulate these types of extreme conditions [4]. WNTR, a Water Network Tool for Resilience, is being developed to evaluate resilience while considering extreme conditions. The WNTR API is flexible and allows for changes to the network structure and operations, along with simulation of disruptie events and recovery actions. WNTR is compatible with EPANET inp files [15].

The focus of this paper is on the hydraulics model used in WNTR. The paper covers, in detail, both why certain models were chosen and how they are implemented. First, we describe the basic modeling components including node mass balances, headloss in pipes, head gain in pumps, tank dynamics, and valve and control operations.

Second, we review several pressure-dependent demand models and present the one used in WNTR. In extreme conditions, the pressures throughout a water network are likely to drop significantly. Common demand driven simulators that assume consumers can always receive their requested demand are not suitable for these scenarios. Thus, WNTR uses a pressure-dependent demand model to more realistically predict demands. In a pressure-dependent demand model, the actual amount of water delivered to consumers depends on the pressures in the network, so the network pressures, network flow rates, and actual delivered demands must be solved for simultaneously.

Finally, we review several pipe leak models and present the one used in WNTR. Pipe leaks are common occurrences due to both pipe deterioration and natural disasters. Additionally, leaks can cause large changes in network hydraulics, so they are modeled explicitly in WNTR.

All model components other than pressure-dependent demand and pipe leaks were taken

*Purdue University School of Chemical Engineering, bynumm@purdue.edu
†Sandia National Laboratories, kaklise@sandia.gov
‡Purdue University School of Chemical Engineering, lairdc@purdue.edu
§Environmental Protection Agency, murray.regan@epa.gov
¶Purdue University School of Chemical Engineering, setha@purdue.edu
‖Sandia National Laboratories, jdsiiro@sandia.gov

from EPANET [15]. EPANET is a water network simulator for both network hydraulics and water quality. EPANET was developed by Lewis Rossman at the Environmental Protection Agency. WNTR does not use EPANET as the hydraulics simulator because we want WNTR to be purely a Python package. Python is an open-source, flexible, high-level language that is easy to use an modify. Although EPANET is not used as the hydraulic simulator, WNTR utilizes the same input file format used by EPANET.

2. Hydraulic Model Components. A hydraulic model represents a water network as nodes connected by links. Nodes include junctions, tanks, reservoirs, and leaks, and links include pipes, pumps, and valves. A hydraulic model consists of this network configuration along with the laws of physics that govern the flow of water throughout the network. A hydraulic simulator uses a hydraulic model to compute the pressures and flow rates throughout the network. This section reviews the hydraulic model components of WNTR.

2.1. Mass Balances at Nodes. WNTR uses the same mass balance equations as EPANET [15]. Conservation of mass (and the assumption of constant density) requires

$$\sum_{p \in P_n} q_{p,n} - D_n^{\text{act}} = 0 \qquad \forall n \in N \tag{2.1}$$

where P_n is the set of pipes connected to node n, $q_{p,n}$ (m^3/s) is the volumetric flow rate of water into node n from pipe p, D_n^{act} (m^3/s) is the actual volumetric demand out of node n, and N is the set of all nodes. If water is flowing out of node n and into pipe p, then $q_{p,n}$ is negative. Otherwise, it is positive.

2.2. Headloss in Pipes. The headloss formula used in WNTR is the Hazen-Williams formula [15]:

$$H_{n_j} - H_{n_i} = h_L = 10.667 C^{-1.852} d^{-4.871} L q^{1.852} \tag{2.2}$$

where h_L is the headloss in the pipe in meters, C is the Hazen-Williams roughness coefficient (unitless), d is the pipe diameter in meters, L is the pipe length in meters, and q is the flow rate of water in the pipe in cubic meters per second. H_{n_j} is the head (meters) at the starting node, and H_{n_i} is the head (meters) at the ending node.

The flowrate in a pipe is positive if water is flowing from the starting node to the ending node and negative if water is flowing from the ending node to the starting node. However, Equation 2.2 is not valid for negative flowrates. Therefore, WNTR uses a reformulation of this constraint:

$$h_L = \begin{cases} -10.667 C^{-1.852} d^{-4.871} L |q|^{1.852} & q < 0 \quad \text{(2.3a)} \\ 10.667 C^{-1.852} d^{-4.871} L |q|^{1.852} & q \geq 0 \quad \text{(2.3b)} \end{cases}$$

Equation 2.3 is symmetric across the origin and valid for any q. Thus, this equation can be used for flow in either direction. However, the derivative with respect to q at $q = 0$ is 0. In certain scenarios, this can cause the Jacobian of the set of hydraulic equations to become singular (when $q = 0$). Therefore, WNTR uses a modified Hazen-Williams formula by default. The modified Hazen-Williams formula splits the domain of q into six segments

to create a piecewise function as presented in Equation 2.4.

$$\frac{h_L}{k} = \begin{cases} -|q|^{1.852} & q < -q_2 \quad (2.4a) \\ -(a|q|^3 + b|q|^2 + c|q| + d) & -q_2 \leq q \leq -q_1 \quad (2.4b) \\ -m|q| & -q_1 < q \leq 0 \quad (2.4c) \\ m|q| & 0 < q < q_1 \quad (2.4d) \\ a|q|^3 + b|q|^2 + c|q| + d & q_1 \leq q \leq q_2 \quad (2.4e) \\ |q|^{1.852} & q_2 < q \quad (2.4f) \end{cases}$$

where m, q_1, and q_2 are appropriate constants and

$$k = 10.667C^{-1.852}d^{-4.871}L \tag{2.5}$$

The result is that flow can be in either direction and the derivative with respect to q is non-zero at all values of q. Equations 2.4b and 2.4e function to smooth the transition from Equation 2.4a to 2.4c and from Equation 2.4d to 2.4f, with coefficients chosen so that both function and gradient values are continuous at $-q_2$, $-q_1$, q_1, and q_2. Appendix A describes this technique in detail. Figures 2.1 and 2.2 compare the Hazen-Williams and modified Hazen-Williams curves, with $m = 0.01$ m$^{2.556}$/s$^{0.852}$, $C = 100$, $d = 0.5$ m, and $L = 200$ m. The figures show that the two formulas are essentially indistinguishable.

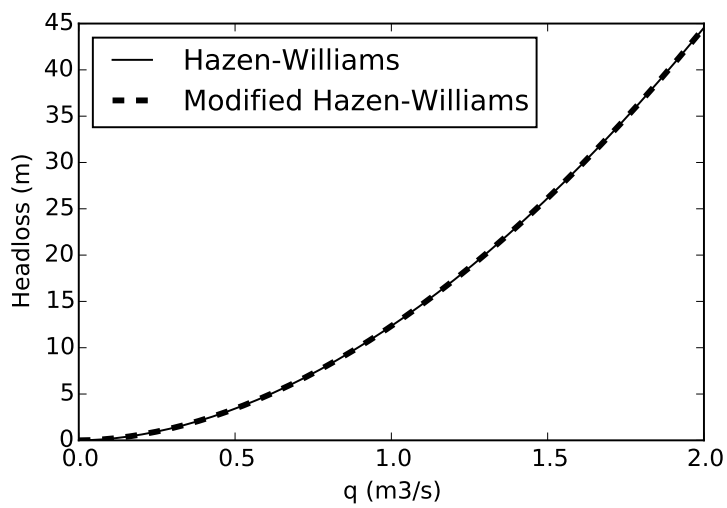


Fig. 2.1: A comparison of the Hazen-Williams formula and the Modified Hazen-Williams formula.

WNTR can support multiple headloss relationships. Future work will include adding other commonly-used formulas such as the Chezy-Manning formula [15] and the Darcy-Weisbach formula [15].

2.3. Pumps. WNTR treats pumps as links that increase the head from the start node to the end node. The gain in head provided by a pump is a function of the flow rate inside the link, and, as in EPANET [15], pump curves are used to describe this relationship. WNTR (and EPANET [15]) represent pump curves as a series of points from the head vs. flow curve. WNTR currently supports single point and three point pump curves. Additionally,

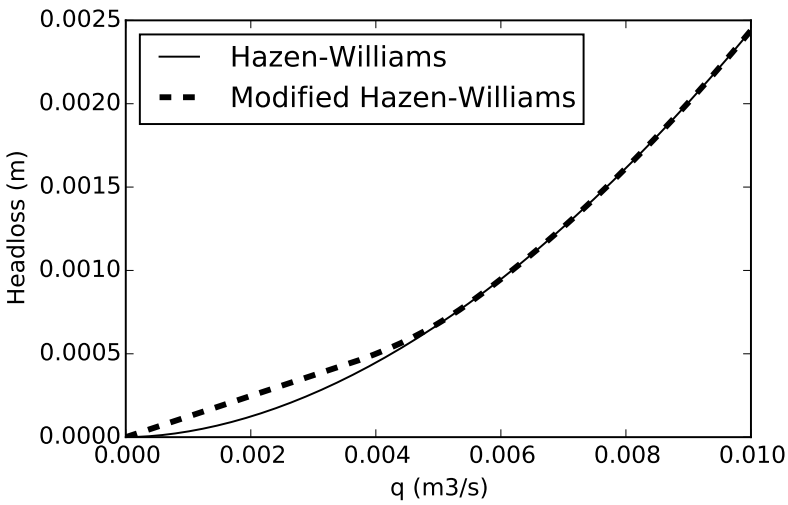


Fig. 2.2: Enlarged view of Figure 2.1.

WNTR supports constant power pumps (pumps that supply a constant power, or energy per unit time, to the fluid). Future work will include support for multi-point curves and variable speed pump curves where the pump curve changes based on speed settings.

2.4. Tanks. WNTR assumes tanks are cylindrical. A mass balance for a cylindrical tank results in

$$\frac{dL}{dt} = \frac{4}{\pi d^2} Q^{\text{net}} \tag{2.6}$$

where L is the tank level in meters, d is the tank diameter in meters, t is time in seconds, and Q^{net} is the net inflow of water into the tank in m^3/s. Because the tank level is proportional to the tank head, $\frac{dL}{dt}$ may be replaced with $\frac{dH}{dt}$, where H is the tank head in meters. WNTR discretizes Equation 2.6 using Explicit Euler to capture the tank level at specific time points and to support solution with a nonlinear programming (NLP) solver. Discretizing with the hydraulic time step as the stepsize results in

$$H_t - H_{t-1} = \frac{4}{\pi d^2} Q^{\text{net}}_{t-1} \triangle t \tag{2.7}$$

where H_t is the tank head at the current time step and H_{t-1} is the tank head at the previous time step. Future work will include support for non-cylindrical tanks via volume curves.

2.5. Valves. Currently, WNTR supports check valves and pressure-reducing valves. Here is a list of valves that may be supported in the future:

- Pressure Sustaining Valve (PSV): Maintain pressure setting upstream.
- Pressure Breaker Valve (PBV): Force specified head loss.
- Flow Control valve (FCV): Limit flow to a specified amount.
- Throttle Control Valve (TCV): Control head-loss coefficient by changing setting.
- General Purpose Valve (GPV): User provided head-loss model.

2.6. Time Based Controls. WNTR has time-based controls for opening and closing links implemented using the same approach used by EPANET [15]. Links may be opened

or closed at specified times during the simulation. In the future, WNTR will also support time-based control of pump speed settings and valve settings.

2.7. Conditional Controls. WNTR supports conditional controls using the same approach used by EPANET [15]. Links are opened or closed based on junction pressures or tank levels. EPANET implements an adaptive step approach that identifies conditional action times and inserts additional hydraulic time steps at these times. WNTR only supports fixed-time discretization. Future work will include support for a similar adaptive step approach and for conditional control of pump speed settings and control valve settings.

3. Pressure-Dependent Demand. Common demand-driven simulators, such as EPANET [15], assume consumers can always receive their requested demand even in pressure-deficient conditions. In scenarios with power outages or pipe leaks, pressure-deficient conditions are likely to occur. Thus, WNTR uses a pressure-dependent demand model in which the actual delivered demands depend on network pressures. The actual demands are solved for simultaneously with the network pressures and flow rates.

3.1. Review. Gupta and Bhave[9] review several pressure-dependent demand models. The first was proposed by Goulter and Coals[8] and Su et al.[16]:

$$D^{\text{act}} = \begin{cases} 0 & P \leq P^{\min} \\ D^{\text{des}} & P \geq P^{\min} \end{cases} \tag{3.1}$$

where D^{act} is the actual demand, D^{des} is the desired demand, P is the pressure, and $P^{\min}$ is the pressure below which the consumer cannot receive any water. The primary downside to this model is that it does not allow for partial flow. Reddy and Elango[14] used a model that does allow partial flow:

$$D^{\text{act}} = S(P - P^{\min})^{0.5} \tag{3.2}$$

where S is a node specific parameter. However, this model allows unlimited flow, making it more applicable to uncontrolled flows such as leaks. Germanopoulos[5] proposed a model of an entirely different form:

$$D^{\text{act}} = D^{\text{des}}(1 - 10^{-c\frac{P-P^{\min}}{P^{\text{nom}}-P^{\min}}}) \tag{3.3}$$

where c is a positive, node-specific constant and P^{nom} is the pressure above which the consumer should receive the desired demand. The upper limit of D^{act} in this equation is D^{des} as we would expect in reality. A significant computational benefit of this model is that it is not a piecewise function. The primary problem is that the functional form does not agree with Torricelli's Law.

Torricelli's Law is a special case of the Bernoulli Equation and states that the flow rate through an orifice is proportional to the square root of the pressure difference across the orifice. If the outlet pressure is atmospheric pressure, then this can be stated as:

$$D \propto \sqrt{h_g} \tag{3.4}$$

where h_g is the gauge pressure head at the inlet.

Wagner et al.[19] proposed a piecewise function that allows partial flow, agrees with Torricelli's Law, and limits the maximum flow:

$$D^{\text{act}} = \begin{cases} 0 & P \leq P^{\min} \\ D^{\text{des}}(\frac{P-P^{\min}}{P^{\text{nom}}-P^{\min}})^{\frac{1}{2}} & P^{\min} \leq P \leq P^{\text{nom}} \\ D^{\text{des}} & P \geq P^{\text{nom}} \end{cases} \tag{3.5}$$

Giustolisi and Walski[7] extend this model for multilevel orifices (e.g., multiple story buildings have faucets at multiple elevations) and explain the need to cap the demand at the requested demand. Above the nominal pressure, the customer controls the flow rate with a faucet.

Wu et al.[20] used a model similar to that proposed by Wagner et al.[19]. Wu et al.[20] used both a nominal pressure and a threshold pressure (P^t):

$$D^{\text{act}} = \begin{cases} 0 & P \leq 0 \\ D^{\text{des}}(\frac{P}{P^{\text{nom}}})^{\alpha} & 0 \leq P \leq P^t \\ D^{\text{des}}(\frac{P^t}{P^{\text{nom}}}) & P \geq P^t \end{cases} \tag{3.6}$$

This allows the actual demand to be higher than the requested demand but still places an upper limit on the actual demand. Note that the exponent, α, is not limited to 0.5. However, in all of their example applications, Wu et al.[20] use a value of 0.5.

3.2. Current Implementation. WNTR uses the pressure-dependent demand (PDD) model proposed by Wagner et al.[19] (Equation 3.5) where P^{min} and P^{nom} must be specified by the user. The reasons for choosing this model are listed below:

- It is commonly used [19, 7, 9, 13, 12, 17].
- The model aligns with what we expect in reality: Below a minimum pressure, the consumer will not get any water; Above a nominal pressure, the consumer will use the desired amount of water and close the faucet; Between the minimum and nominal pressures, the consumer will receive partial flow.
- The square root form of the equation for partial flow agrees with Torricelli's Law [6].

Because the pressure-dependent demand model does not have a continuous derivative, a smoothing technique is used to assist the solver as done for the Hazen-Williams formula above. Appendix A describes the procedure in detail. Figure 3.1 compares the original PDD model to the PDD model with smoothing ($P^{\text{min}} = 5$ psig, $P^{\text{nom}} = 25$ psig, and $D^{\text{des}} = 50$ gpm).

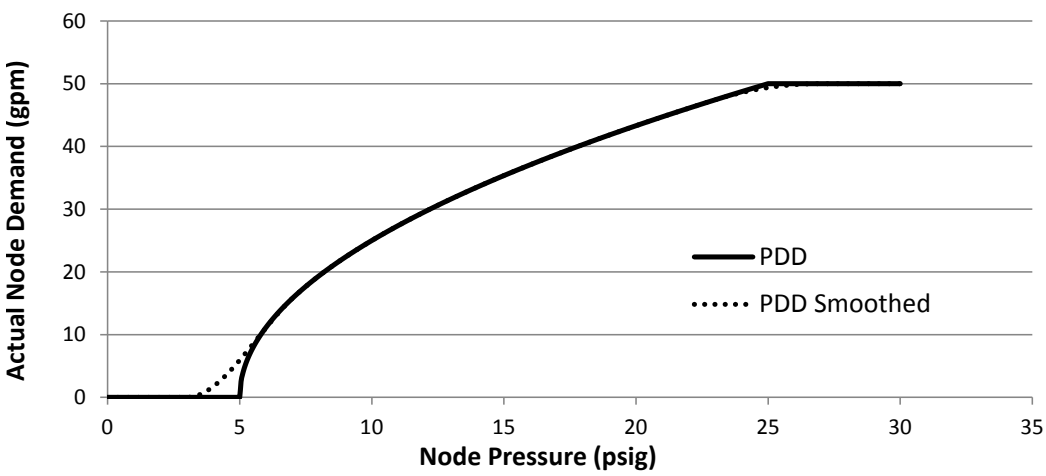


Fig. 3.1: A comparison of the original PDD model and the smoothed PDD model.

To increase flexibility, future work includes implementing a more general form of the PDD model where the exponent is $\frac{1}{n}$ and n is any positive integer.

4. Pipe Leaks. Pipe leaks are common occurences in water networks, especially after a natural disaster. Pipe leaks can drastically change network hydraulics and, therefore, must be modeled. In this section, we review several pipe leak models and then present the model used in WNTR.

4.1. Review. According to Crowl and Louvar[3], a leak through a hole may be modeled as an orifice:

$$D^{\text{leak}} = AC_d\sqrt{2\rho P_g} \tag{4.1}$$

where D^{leak} is the mass flow rate of fluid through the hole, A is the area of the hole, C_d is the discharge coefficient, ρ is the density of the fluid, and P_g is the gauge pressure inside the process equipment. The derivation of Equation 4.1 assumes that the pressure outside the pipe is atmospheric pressure. If this assumption is not valid, P_g should be replaced with $\triangle P$, the absolute pressure inside the pipe minus the absolute pressure outside the pipe. The discharge coefficient approaches 0.61 for sharp-edged orifices and Reynolds numbers larger than 30,000. It approaches unity for well-rounded nozzles. Crowl and Louvar state that a value of 1.0 should be used when the discharge coefficient is unknown in order to maximize the flow.

The difficulty with Equation 4.1 is that C_d and A may vary with pressure. Giustolisi and Walski[7] model leaks as

$$D^{\text{leak}} = \begin{cases} \beta L P^{\gamma} & P \geq 0 \\ 0 & P < 0 \end{cases} \tag{4.2}$$

where L is the pipe length, P is the pressure inside the pipe, and β and γ are parameters. Parameter values for γ are taken from other sources. For more information, see page 365 of Giustolisi and Walski[7].

Cassa et al.[2] investigate the change in leak size (or leak area) due to the pressure in the pipe. They report that the leak flow rate is proportional to h^{N1}, where h is the pressure head and $N1$ varies between 0.5 and 2.79 with a mean of 1.15. They seek to explain this dependence (at least partially) with a relationship between area and pressure. They find that hole size is linearly dependent on pressure, giving:

$$A = mP + A_0 \tag{4.3}$$

where A_0 is the area of the hole without any pressure in the pipe and m is the slope of the area-pressure relationship. Substituting Equation 4.3 into Equation 4.1 results in

$$D^{\text{leak}} = C_d\sqrt{2\rho}(mP_g^{1.5} + A_0P_g^{0.5}) \tag{4.4}$$

Additionally, Cassa et al.[2] found that round holes had small slopes, indicating that the hole size only increased slightly with pressure. However, longitudinal and circumferential cracks had relatively large slopes.

Lambert[11] gave an overview of pressure-leakage relationships. Again, $D^{\text{leak}} \propto P^{N1}$. A list of the major points is below.

- The discharge coefficient, C_d, varies with the Reynolds number. However, for fully turbulent flow, $C_d \approx 0.75$.
- For longitudinal splits, $D^{\text{leak}} \propto P^{1.5}$.
- For longitudinal and radial splits, $D^{\text{leak}} \propto P^{2.5}$.
- In one study in Japan, holes were drilled in metal pipes to represent leaks, and the pipes were placed in sand or water. $N1$ was found to be between 0.36 and 0.7.

- A study in the UK found $N1$ to be near 0.5 for metal pipes and 1.5 for plastic pipes.
- Ashcroft and Taylor[1] performed tests on 22 mm Class D polyethylene pipe. For a 10 mm slit, $N1 \approx 1.39 - 1.72$. For a 20 mm slit, $N1 \approx 1.23 - 1.97$. The average was 1.52.
- Ogura tested sectors of Japanese water distribution systems. Most of the distribution systems had metal mains. Ogura found $N1$ values between 0.65 and 2.12 with an average of 1.15. This average ($N1 = 1.15$) was used as the Japanese standard from approximately 1980 to at least 2000. Yeung later showed that Ogura's results may have been skewed high due to variation in the discharge coefficient.

Lambert[11] concludes that $N1$ should be around 1.5 for "small 'background' leaks", 1.5 or higher for larger leaks from plastic pipes, 0.5 for larger leaks out of metal pipes, and 1.0 when lacking information on pipe material or type of leak.

4.2. Current Implementation. In light of the above review, the primary factors affecting the discharge coefficient and the exponent on pressure are flow regime, pipe material, and orifice shape (longitudinal, round, circumferential). The current implementation in WNTR uses Equation 4.5 as the default pipe leak model, which is a more general form of Equation 4.1.

$$D^{\text{leak}} = C_d A_0 \sqrt{2\rho} P_g^{\alpha} \tag{4.5}$$

The default discharge coefficient is 0.75 (assuming turbulent flow [11]), but the user may specify other values if needed. The value of α is set to 0.5 (assuming large leaks out of steel pipes [11]). The user is required to specify the orifice area and the pipe for which the leak occurs. The current implementation in WNTR simply adds a node to the network at the location of the leak (halfway down the specified pipe) and sets the demand based on Equation 4.5.

In the future, more detail will be added to the model to account for different types of pipe material, different types of leaks, and different flow regimes. For example, we can set α to different values depending on the pipe material or implement a variable area model (a more general form of Equation 4.4) that can be used instead of the default if m is provided:

$$D^{\text{leak}} = C_d \sqrt{2\rho}(A_0 P_g^{\gamma} + m P_g^{\gamma+1}) \tag{4.6}$$

We will also allow for user-defined location of leaks (along the specified pipe) in the future.

5. Modeling Tools and Solvers. WNTR is a python package. Pyomo [10] is used as the modeling language and Ipopt [18] is used as the solver. Pyomo is a python-based optimization modeling language. Pyomo was used for several reasons. First, Pyomo has extensive modeling capabilities, such as built in tools for piecewise constraints, absolute values, etc. Second, Pyomo is open source. Finally, because Pyomo is python based, the framework is very flexible. This is important for WNTR because the set of hydraulic equations has to be solved many times sequentially, with the activation of controls, re-initialization of the problem, and many other operations between each time step. For these reasons, Pyomo was favorable despite its design for optimization. Ipopt was chosen both because it is open source and because it can solve large scale problems efficiently. Additionally, Pyomo has an interface to Ipopt. An additional benefit to using Pyomo and Ipopt is that an objective function can easily be added to the problem if desired.

6. Conclusions. WNTR, a Water Network Tool for Resilience is a tool for evaluating and improving resilience of water networks to adverse events such as terrorist attacks, earthquakes, power outages, etc. Hydraulic models used in WNTR were reviewed in order to

present the current and future capabilites of this tool. The key hydraulic model components are node mass balances, headloss in pipes, pump head gain, tank dynamics, valve operations, controls, pressure dependent demand, and pipe leaks. Pressure-dependent demand is needed to realistically predict demand in pressure deficient conditions. The pressure-dependent demand model used is that proposed by Wagner et al.[19] because it allows partial flow, it places an upper limit on the demand, and it agrees with Torricelli's Law. A smoothing technique is used to ensure continuity of the PDD function and its derivative. A review of pipe leak models showed that most models have a similar form. The differences appear in the values of the discharge coefficient, the leak area, and the exponent on the pressure. The primary factors affecting these values are flow regime, pipe material, and leak shape. Additionally, the leak area may be dependent on pressure. WNTR currently uses a simple model (Equation 4.5) with a fixed area and an exponent of 0.5. However, more detail will be added to the model to account for these factors in the future.

Disclaimer. The U.S. Environmental Protection Agency (EPA) through its Office of Research and Development funded and collaborated in the research described here under an Interagency Agreement with the Department of Energy's Sandia National Laboratories. It has been subjected to the Agency's review and has been approved for publication. Note that approval does not signify that the contents necessarily reflect the views of the Agency. Mention of trade names products, or services does not convey official EPA approval, endorsement, or recommendation.

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

A. Smoothing Piecewise Functions. This appendix describes the smoothing technique used for the Hazen-Williams headloss formula and for the pressure-dependent demand model. Consider the following piecewise function:

$$y = \begin{cases} f_1(x) & x \leq x_1 \quad \text{(A.1a)} \\ f_2(x) & x > x_1 \quad \text{(A.1b)} \end{cases}$$

where x_1 is a constant. Consider the case where the piecewise function is continuous, i.e.,

$$f_1(x_1) = f_2(x_1) \tag{A.2}$$

but the derivative is discontinuous, i.e.,

$$\left.\frac{df_1}{dx}\right|_{x=x_1} \neq \left.\frac{df_2}{dx}\right|_{x=x_1} \tag{A.3}$$

Because a discontinuous derivative can cause problems for many solvers, we insert a third order polynomial between f_1 and f_2:

$$y = \begin{cases} f_1(x) & x \leq (x_1 - \delta_1) \quad \text{(A.4a)} \\ f_s(x) = ax^3 + bx^2 + cx + d & (x_1 - \delta_1) < x < (x_1 + \delta_2) \quad \text{(A.4b)} \\ f_2(x) & x \geq (x_1 + \delta_2) \quad \text{(A.4c)} \end{cases}$$

We solve for the coefficients of Equation A.4b such that both the function and its derivative are continuous:

$$0 = f_1(x_1 - \delta_1) - f_s(x_1 - \delta_1) \tag{A.5}$$

$$0 = f_2(x_1 + \delta_2) - f_s(x_1 + \delta_2) \tag{A.6}$$

$$0 = f_1'(x_1 - \delta_1) - f_s'(x_1 - \delta_1) \tag{A.7}$$

$$0 = f_2'(x_1 + \delta_2) - f_s'(x_1 + \delta_2) \tag{A.8}$$

REFERENCES

[1] A. Ashcroft and D. Taylor, *Ups and downs of flow and pressure*, Surveyor, 162 (1983), pp. 16–18.

[2] A. Cassa, J. van Zyl, and R. Laubscher, *A numerical investigation into the effect of pressure on holes and cracks in water supply pipes*, Urban Water Journal, 7 (2010), pp. 109–120.

[3] D. A. Crowl and J. F. Louvar, *Chemical Process Safety: Fundamentals with Applications*, Prentice Hall, 2002.

[4] U. S. EPA, *Systems measures of water distribution system resilience*, Tech. Rep. EPA/600/R-14/383, U.S. Environmental Protection Agency, Washington, DC, 2015.

[5] G. Germanopoulos, *A technical note on the inclusion of pressure dependent demand and leakage terms in water supply network models*, Civil Engineering Systems, 2 (1985), pp. 171–179.

[6] O. Giustolisi and T. Walski, *Demand components in water distribution network analysis*, Journal of Water Resources Planning and Management, 138 (2011), pp. 356–367.

[7] O. Giustolisi and T. Walski, *Demand components in water distribution network analysis*, Journal of Water Resources Planning and Management, 138 (2012), pp. 356–367.

[8] I. C. Goulter and A. Coals, *Quantitative approaches to reliability assessment in pipe networks*, Journal of Transportation Engineering, (1986).

[9] R. Gupta and P. R. Bhave, *Comparison of methods for predicting deficient-network performance*, Journal of Water Resources Planning and Management, 122 (1996), pp. 214–217.

[10] W. Hart, C. Laird, J. Watson, and D. Woodruff, *Pyomo: Optimization Modeling in Python*, vol. 67, Springer Verlag, 2012.

[11] A. Lambert, *What do we know about pressure-leakage relationships in distribution systems*, Proceedings of the IWA Conference: Systems Approach to Leakage Control and Water Distribution Systems Management, (2001).

[12] D. Laucelli, L. Berardi, and O. Giustolisi, *Assessing climate change and asset deterioration impacts on water distribution networks: demand-driven or pressure-driven network modeling?*, Environmental Modelling & Software, 37 (2012), pp. 206–216.

[13] A. Ostfeld, D. Kogan, and U. Shamir, *Reliability simulation of water distribution systems–single and multiquality*, Urban Water, 4 (2002), pp. 53–61.

[14] L. S. Reddy and K. Elango, *Analysis of water distribution networks with head-dependent outlets*, Civil Engineering Systems, 6 (1989), pp. 102–110.

[15] L. A. Rossman, *Epanet 2: Users manual*, (2000).

[16] Y.-C. Su, L. W. Mays, N. Duan, and K. E. Lansey, *Reliability-based optimization model for water distribution systems*, Journal of Hydraulic Engineering, 113 (1987), pp. 1539–1556.

[17] N. Trifunovic, *Pattern Recognition for Reliability Assessment of Water Distribution Networks*, PhD thesis, University of Belgrade, 2012.

[18] A. Wächter and L. Biegler, *On the Implementation of an Interior-Point Filter Line-Search Algorithm for Large-Scale Nonlinear Programming*, Mathematical Programming, 106 (2006), pp. 25–57.

[19] J. M. Wagner, U. Shamir, and D. H. Marks, *Water distribution reliability: Simulation methods*, Journal of Water Resources Planning and Management, 114 (1988), pp. 276–294.

[20] Z. Y. Wu, R. H. Wang, T. M. Walski, S. Y. Yang, D. Bowdler, and C. C. Baggett, *Extended global-gradient algorithm for pressure-dependent water distribution analysis*, Journal of Water Resources Planning and Management, 135 (2009), pp. 13–22.

ASSESSING THE ECONOMIC VALUE OF GRID-SCALE ENERGY STORAGE SYSTEMS FOR POWER SYSTEM EXPANSION PLANNING

RODERICK S. GO*, FRANCISCO D. MUÑOZ†, AND JEAN-PAUL WATSON‡

Abstract. With increasing renewable energy investment, there is growing interest in both grid-scale energy storage systems (ESS) as part of the transmission expansion planning problem to better utilize and deliver this power to consumers. Furthermore, environmental regulations, such as Renewable Portfolio Standards (RPSs), will influence the types of investment and operations decisions that are economically feasible. Using a stochastic framework, we develop a unified generation, ESS, and transmission expansion planning model formulated as a two-stage, Mixed Integer Linear Problem (MILP). This formulation allows us to explore the economic value of ESS as a transmission asset in providing bulk energy management services to increase renewable penetration.

1. Introduction. The electricity industry is in the midst of significant transformation related to increasing renewable generation. Several studies have documented the challenges associated with achieving significant levels of renewable generation, such as resource intermittency, transmission expansion, and regulatory uncertainty [7], [17], [18], [24]. While these studies find that 20% is achievable without drastic changes, to generate the majority of our electricity from renewable resources requires considering new means of system expansion planning, scheduling, and regulation. System operators in particular are faced with significant uncertainties for the future in terms of demand growth, new generation capacity, fuel costs, and regulations. As such, planners in both vertically integrated and decentralized utility must be able to effectively model and study various technical, cost, and policy regimes to make investment decisions to yield the lowest cost.

A key element of the transformation of the electricity industry is regulatory policies to promote change in generation technologies. RPS requirements are popular market mechanisms to incentivize generators to produce a minimum fraction of electricity from renewable resources. The market provides flexibility for generators to purchase or sell Renewable Energy Certificates (REC) to meet their RPS requirements. Previous studies show how different implementations of state- v. federal-level RPS requirements can change both generation and transmission investment decisions [21], [28]. On the other hand, Muñoz, et al. show how different approximations of the network model itself, such as ignoring transmission congestion, can also lead to biased investment decisions to meet RPS requirements for the lowest cost [19]. To meet RPS requirements for the lowest cost, system planners must be able to effectively model and maximize the benefits of available renewable resources. Using a co-optimization model, transmission planners can anticipate and promote investment in specific locations, proactively building new transmission or reinforcing existing corridors.

While regulatory policies such as RPS requirements encourage new generation investment and generation, planners and operators must take into account the inherent intermittency of the most popular renewable resources: wind and solar. Intermittency is a cause of significant concern, because it reduces the value of these new investments and requires that operators have enough reserve capacity to maintain a balance between supply and demand through the day. Previous studies, such as [20], have studied the impact and capacity value of renewable resources at high penetration levels, finding a decreasing capacity contribution as renewable penetration increases. A significant factor in this decreasing value of renewables is the need to maintain significant thermal reserve capacity on standby to mitigate

*Johns Hopkins University, rgo@jhu.edu
†Universidad Adolfo Ibáñez and Sandia National Laboratories, fdmunoz@uai.cl
‡Sandia National Laboratories, jwatson@sandia.gov

the risk of resource intermittency. While some of the variability in wind can be mitigated by building generation at multiple locations, the issue of time-variability remains for solar. While significant solar penetration can effectively eliminate the mid-day peak, as the sun sets, the system must quickly rebalance by dispatching other resources to meet the evening peak. The California Independent System Operator showed this pattern with its infamous "duck curve" graph.

As such, there is significant interest in providing system flexibility–through market mechanisms, regulatory requirements, or technical improvements. One area of intense commercial and research interest is in energy storage, which can allow system operators to decouple supply and demand as well as manage network congestion and other grid services. For this reason, utilities such as PJM Interconnection and the California Public Utilities Commission believe there is an economic value to energy storage systems (ESS) as a transmission asset and have begun investing in bulk energy storage systems at key points in their networks [1], [12].

Previous studies have attempted to optimally allocate and operate ESS as a network asset. Early work also focused on the ability of ESS to increase network reliability by providing backup power for systems with high renewable penetration [3] as well as coordinated control to correct for both short- and long-term power overloads in faulted network conditions [31]. Pandžić, et al. created a novel algorithm in [23] to quickly allocate storage while performing a sensitivity analysis on capital costs. Faghih, et al. also studied the economic value of ESS in the presence of ramp constraints, showing that the value of ESS is a non-decreasing function of price volatility, though the value of storage decreases significantly at high capacities due to limited system ramping [8]. Further, Li and Hedman showed that ESS can also lower switching and re-dispatch costs after network faults in N-1 contingency cases [16]. However, they also showed that ESS could not completely offset the need for transmission investments to maintain system security.

Various studies have also modeled economically efficient investment in ESS as part of transmission expansion planning model [2], [9], [11]. In this context, ESS provides bulk energy services for the network, such as load shifting and peak shaving. We extend this work by assessing the economic interaction between investments in generation, energy storage, and transmission on a simulated network under specific regulatory scenarios using a two-stage, stochastic transmission expansion planning model. This study gives us insight into the value of energy storage in relation to other available resources in increasing resource adequacy and utilization and regulatory compliance. This expansion planning model can be scaled up for use with realistic systems, but such work is beyond the scope of this paper.

2. Nomenclature. In this section, we present the nomenclature for the stochastic transmission expansion planning model with storage.

2.1. Sets and Indices.

t	Hours in day
b,b'	Buses on lines in network
g	Generators
$n(g)$	New generators
$r(n)$	New renewable generators
l	Transmission lines (existing and new)

2.2. Parameters.

C_n^{inv}	Annualized investment cost for new generators [$MM/MW-yr]
$C_g^{om,f}$	Fixed O&M cost for generators [$/MW installed]

$C_g^{om,v}$	Variable O&M cost for generators [$/MWh]
C_g^{fuel}	Fuel cost for generators [$/MMBtu]
HR_g	Heat rate for thermal generators [MMBtu/MWh]
P_g^{max}	Upper bound of power generation for generator g
C^{es}	Annualized investment cost for energy storage for ESS [$MM/MWh-yr]
C^{pc}	Annualized investment cost for power conversion for ESS [$MM/MW-yr]
$C^{om,es}$	Fixed O&M cost for ESS [$MM/MW installed]
C^d	Discharge (variable O&M) cost for ESS [$MM/MWh discharged]
η^c	Charging efficiency for ESS
η^d	Discharging efficiency for ESS
C_l^{inv}	Annualized investment cost for new transmission line [$MM/yr]
F_l^{max}	Line real power flow capacity [MW]
B_l	Susceptance on line l
$P_{b,t}^d$	Demand at bus b in time t [MW]
C^{ue}	Cost penalty for unserved energy [$/MWh]
REC	Price of Renewable Energy Certificate [$/MWh]
RPS	Renewable Portfolio Standard requirement
CAP_r^{min}	Renewable Capacity minimum requirement
H_t	Probabilistic weighting of each hour in the day
M	Big M parameter for disjunctive constraints

2.3. Variables.

2.3.1. Investment Variables.

k_b^{es}	Continuous investment for energy storage component of ESS at bus b
k_b^{pc}	Continuous investment for power conversion component of ESS at bus b
$x_{n,b}^g$	Integer investment for generator type n at bus b
x_l	Binary investment for new transmission lines l

2.3.2. Operations Variables.

$p_{g,b,t}$	Dispatch of generators at bus b in time t [MW]
$p_{b,t}^{ue}$	Load shed at bus b in time t [MW]
$f_{l,t}$	Flow on line l in time t [MW]
$\theta_{b,t}$	Phase angle at bus b in time t [radians]
$s_{b,t}$	Energy level in ESS at bus b in time t [MWh]
$r_{b,t}^c$	Charging of ESS at bus b in time t [MW]
$r_{b,t}^d$	Discharging of ESS at bus b in time t [MW]
$b_{b,t}^{rc}$	Binary variable indicating that ESS is charging
$b_{b,t}^{rd}$	Binary variable indicating that ESS is discharging

3. Model Formulation. In this section, we present the cost-minimization problem formulation for a centralized planner, allowing investment into thermal and renewable generators, ESS, and transmission lines. The problem can be thought of as a two-stage problem, with the first stage being the investment model–where all investment decisions are made–and the second stage being based on an economic dispatch model–where the investment decisions from the first stage are tested against different scenario days.

In this formulation, we use an idealized representation of lossy ESS, with the energy storage (i.e. kWh stored) and power conversion (i.e. kW charged or discharged) components decoupled, allowing the model to decide the optimal power-to-energy ratio for the system, similar to the formulation in [23]. This formulation can easily be adjusted to take into consideration fixed parameters based on a portfolio of real ESS, such as pumped-hydro storage, compressed air storage, or batteries. With this consideration, the first stage costs (C^{fs}) is to minimize investment cost over new generators, transmission lines, and ESS capacity, while the second stage costs (C^{ss}) include operational cost over all assets, as well as considering RPS compliance and unserved energy penalties.

$$\min \; C^{fs} + C^{ss} \tag{3.1}$$

where:

$$C^{fs} = \sum_{n,b} C_n^{inv} x_{n,b}^g + \sum_l C_l^{inv} x_l + \sum_b C^{inv,es} k_b^{es} + C^{inv,pc} k_b^{pc} \tag{3.2}$$

$$C^{ss} = \sum_{g,b,t} H_t \left(C_g^f HR_g + C_g^{om,v} \right) p_{g,b,t} + \sum_g C_g^{om,f} P_g^{max} \tag{3.3}$$

$$+ \sum_{b,t} H_t C^d r_{b,t}^d + \sum_b C^{om,es} k_b^{pc} \tag{3.4}$$

$$+ \sum_{b,t} H_t C^{ue} p_{b,t}^{ue} \tag{3.5}$$

$$+ \sum_{b,t} REC \cdot H_t \left(RPS \cdot P_{b,t}^d - \sum_{r(g)} p_{g,b,t} \right) \tag{3.6}$$

Due to the long-term planning nature of this model, more detailed constraints, such as unit commitment variables, forced outage rates, ramp rates, and lower bounds for power generators are not explicitly considered. Previous studies have discussed the tradeoffs associated with excluding different operating constraints in generation planning models [22].

$$\text{s.t.} \quad \sum_{l(:,b)} f_{l,t} - \sum_{l(b,:)} f_{l,t} + p_{g,b,t} + r_{b,t}^d - r_{b,t}^c + p_{b,t}^{ue} = P_{b,t}^d \qquad \forall b,t \tag{3.7}$$

$$p_{g,b,t} \leq P_g^{max} \qquad \forall g,b,t \tag{3.8}$$

$$-F_l^{max} \leq f_{l,t} \leq F_l^{max} \qquad \forall l,t \tag{3.9}$$

$$f_{l,t} = B_l \left(\theta_{b,t} - \theta_{b',t} \right) \qquad \forall l,t \tag{3.10}$$

$$\tag{3.11}$$

For new generators or lines, (3.8)–(3.10) must be modified as disjunctive constraints to incorporate the investment decision:

$$p_{n,b,t} \leq P_g^{max} x_{n,b}^g \qquad \forall n(g),b,t \tag{3.12}$$

$$-F_l^{max} x_l \leq f_{l,t} \leq F_l^{max} x_l \qquad \forall l,t \tag{3.13}$$

$$-M\left(1 - x_l\right) \leq f_{l,t} - B_l \left(\theta_{b,t} - \theta_{b',t} \right) \leq M\left(1 - x_l\right) \qquad \forall l,t \tag{3.14}$$

$$\tag{3.15}$$

Next, we add the storage constraints. Response times for ESS, which may range from milliseconds to many minutes depending on specific technology, are not considered in this

formulation. This is similar to the exclusion of unit commitment constraints in the generation formulation.

$$s_{b,t} = s_{b,t-1} + \Delta t \left(\eta^c r^c_{b,t} - r^d_{b,t}/\eta^d\right) \qquad \forall b, t \tag{3.16}$$

$$s_{b,t} \leq k^{es}_b \qquad \forall b, t \tag{3.17}$$

$$r^c_{b,t} \leq k^{pc}_b \qquad \forall b, t \tag{3.18}$$

$$r^d_{b,t} \leq k^{pc}_b \qquad \forall b, t \tag{3.19}$$

$$r^c_{b,t} \leq M \cdot b^{rc}_{b,t} \qquad \forall b, t \tag{3.20}$$

$$r^d_{b,t} \leq M \cdot b^{rd}_{b,t} \qquad \forall b, t \tag{3.21}$$

$$b^{rc}_{b,t} + b^{rd}_{b,t} \leq 1 \qquad \forall b, t \tag{3.22}$$

We also explicitly prevent ESS at each bus from simultaneously charging and discharging in (3.20)–(3.22). This phenomenon can occur in a model that does not track charging and discharging if there are sufficiently negative locational marginal prices; however, in practice such operational behavior is unrealistic. If this phenomenon were not explicitly prevented, we expect that the model would tend to over-estimate the value of ESS by allowing it to provide an unrealistic increase in system flexibility. This is reflected in a consistent reduction in total system cost and general trend of increased investment into ESS over the same scenarios without simultaneous charging, shown in Table 3.1.

Table 3.1: Change in Total System Cost Due to Simultaneous Charging/Discharging in ESS

	Δ Total Cost [$MM]	Δ ESS Investment [$MM]
0% RPS	0	3.8
25% RPS	0	-11.8
40% RPS	-14	9.8
75% RPS*	-49	9.8

Negative prices are increasingly a reality in power systems, due to regulatory incentives such as Production Tax Credits (PTCs), Feed-In Tariffs (FITs), or in the case of our study, RECs associated with RPS requirements. Due to the non-dispatchability of renewables, in low demand–high sun/wind availability situations it can be cheaper or even profitable to keep renewables online and get credit for renewable generation toward PTCs, FITs, or RECs [5]. While ESS can provide some flexibility in the system by absorbing excess generation (thereby reducing the likelihood of negative prices), in scenarios with significant RPS requirements and very high REC prices, negative locational marginal prices may simply be unavoidable. While previous studies, such as [13] and [14], have used binary variables to track ESS charging and discharging in operational and ESS control models, to the best of our knowledge, there have been no previous studies using binary variables for co-optimization in joint generation, transmission, and ESS expansion planning model.

Finally, we make the problem separable by day for future solution algorithms (e.g. parallel Benders decomposition) by further requiring that the energy storage level (3.16) at the beginning of each day coincide with the energy storage level at the end of the same day:

$$s_{b,1} = s_{b,T} + \Delta t \left(\eta^c r^c_{b,t} - r^d_{b,t}/\eta^d\right) \tag{3.23}$$

A similar formulation to link the first and last hour of each sampled day is presented in [15]. By using this constraint, we avoid linking the otherwise disconnected days from our sampled year.

4. 24-Bus IEEE Reliability Test System with Renewables. Results in this paper are based on the 24-bus IEEE Reliability Test System 1996 (IEEE RTS-96), which includes 24 buses, 34 transmission lines, and 32 existing generators with a total installed capacity of 3,405 MW [4]. The existing generators are separated into nine groups based on fuel and generation technology. Peak system demand is set at 6,000 MW for all tests. A schematic of the system is shown in Figure 4.1. The system is divided into two voltage areas.

Updated costs for existing generators are shown in Table 4.1, based on current fuel and O&M costs from the U.S. Energy Information Agency [26], [27]. Capacities and costs for new generators are compiled from the same data and shown in Table 4.2.

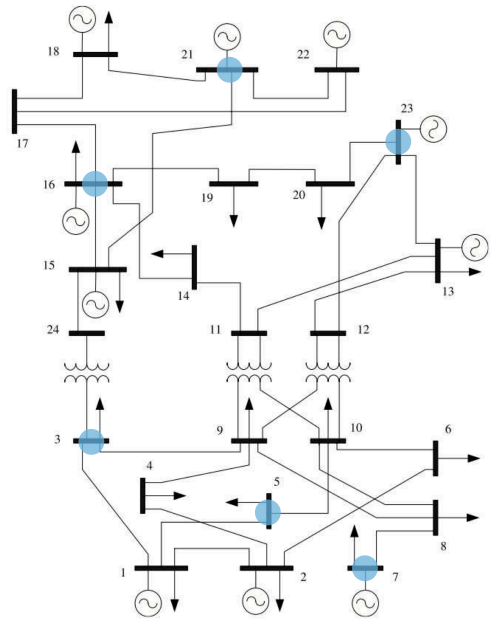


Fig. 4.1: Diagram of 24-bus IEEE Reliability Test System 1996. Six buses with available wind for renewable investment are marked in blue, each with a different wind profile. Adapted from [6].

Investments into new transmission lines are limited to existing corridors. Costs for transmission investments are derived from the prescribed line length from the IEEE RTS-96 and investment cost of \$0.45MM/mile for 138 kV lines and \$0.95MM/mile for 230 kV lines from [25]. Cost information for ESS is based on costs from surveys and previous studies [23], [29]. ESS is assumed to have an 81% roundtrip efficiency. The fixed and variable O&M costs for ESS are set at \$5000/MW and \$5/MWh for all tests, with investment cost into energy storage components at \$50/kWh and power conversion components at \$1000/kW, asssuming a 20 year lifespan.

Table 4.1: Updated Capital and Operating Costs for Existing Generators in the IEEE RTS-96

Gen. Group	P_g^{max}	C_g^{fuel}	$C_g^{om,f}$	$C_g^{om,v}$	HR_g
U12	12	11.47	13,170	3.6	12,000
U20	20	16.53	7,040	15.5	14,500
U50	50	–	14,800	2.65	–
U76	76	2.26	51,400	7.22	12,000
U100	100	11.47	13,170	3.6	10,000
U155	155	2.26	51,400	7.22	9,700
U197	197	11.47	13,170	3.6	9,600
U350	350	2.26	51,400	7.22	9,500
U400	400	0.72	93,300	2.14	10,000

Table 4.2: Capital and Operating Costs for New Generators

Technology	P_n^{max}	C_n^{fuel}	$C_n^{om,f}$	$C_n^{om,v}$	HR_n	C_n^{inv}
CC	620	4.38	13,170	3.6	7.05	0.728
CT	85	4.38	7,040	15.45	10.85	0.51
Nuc.	2,234	0.72	93,280	2.14	9.612	0.385
IGCC	600	2.26	62,250	31.7	8.7	0.295
PV	–	–	24,690	–	–	0.276
Wind	–	–	39,550	–	–	0.183

Annual demand, wind, and solar PV profiles are obtained from Muñoz and Mills [20]. These profiles are based on five years of data, normalized for demand growth to a peak annual demand of 6000 MW. These profiles are then sampled over 10,000 replications, and the best sampled is selected based on minimization of the sum of squared differences of means, variances, and correlations between the original data and the sample.

As shown in Figure 4.1, wind is available in six buses, with wind profiles at each bus having different correlations with respect to demand to simulate local variability in wind resources. Two solar profiles (one for each of the two voltage areas) are available at all 24 buses in the system. The authors of the original data set suggest that a sample size of at least 50 representative days is necessary to accurately reflect the original data. However, for the purposes of this paper, a sample of five representative days is used to make the case studies computationally tractable.

5. Case Studies. Using the model described in Section 3, we perform a sensitivity analysis on key parameters related to energy storage and renewable generation regulations (namely RPS requirements) to understand the impact of ESS on regulatory compliance. All model runs use the Pyomo 4.1 [10] modeling language and associated PySP stochastic programming package [30] and are solved using CPLEX 12.6.2 on a quad-core mobile workstation with 8 GB of RAM. All cases were solved to a 0.5% optimality gap tolerance, unless marked by an asterisk (*).

5.1. Cost Reductions. Table 5.1 summarizes the change in investments for generators, transmission lines, and ESS, and the change in expected operations costs (including RPS non-compliance and load shedding costs) in each RPS requirement scenario.

Table 5.1: Change in Investment and Expected Operations Costs Due to ESS Availability [$MM]

	ESS	Δ Total	Δ Gen.	Δ Lines	Δ O&M
0% RPS	98	-214	-148	-57	-107
25% RPS	181	-294	45	-40	-481
40% RPS	221	-435	3	-79	-580
75% RPS*	324	-820	424	15	-1,573

In each RPS scenario, we see a significant savings in total system cost. For generator investments, we savings in the 0% RPS scenario, but increases in generation investment–namely in renewable generators–due to increased RPS incentive when paired with ESS. In all but the most extreme RPS scenario, we see reductions in transmission line investments as well, with fewer lines built to reinforce corridors with renewable generation. Significant operational savings are seen as well, largely due to reductions in load shedding, RPS non-compliance, with a minority of savings coming from reduced fuel consumption.

5.2. Utilization of ESS to Reduce Unserved Energy. To test the value of storage with respect to reducing unserved energy in the system, we set the value of lost load–the penalty incurred for unserved energy–at $1000/MWh, which is used in many markets. At various RPS levels, we calculate both the expected and peak unserved energy, shown in Table 5.2.

Table 5.2: Expected and Peak Unserved Energy for Each RPS Scenario

	Expected Unserved [MWh/day]		Peak Unserved [MW]		ESS Investment
	No ESS	With ESS	No ESS	With ESS	[$MM]
0% RPS	174	36	182	80	98.2
25% RPS	307	11	280	29	181.4
40% RPS	304	5	285	25	211.4
75% RPS*	368	0	324	0	324.4

As RPS requirements (though not necessarily RPS compliance, as shown in Table 5.3) increases, we see that both expected and peak unserved energy increases for cases without ESS, meaning that it is more economical to accept the $1,000/MWh cost than invest in a new peaking generator. However, the reverse is true for cases with ESS: both expected and peak unserved energy decrease with higher RPS requirements, as there is more ESS capacity to economically absorb the load imbalances in the system. From this, we can conclude that ESS availability is able to offset some investment into expensive, peaking generators by providing some bulk energy services in times of high demand.

5.3. Contribution of ESS to Increasing RPS Compliance. A Renewable Portfolio Standard incentivizes generators to generate a fraction of total energy from renewables using a Renewable Energy Certificate system that sets a price for renewable generation. Currently, different states in the US have RPS requirements as high as 33%, with discussion in California to raise that to 40%. However, the challenge with increasing renewable penetration is the increased volatility of supply: as a result, significant amounts of fast responding reserve capacity and other flexibility mechanisms, such as ESS, must be incorporated to meet demand.

To model RPS compliance, we use the method of Lagrangian relaxation to add the RPS requirement directly to the objective (3.6). With Lagrangian relaxation, under- or over-compliance is penalized or rewarded based on a fixed cost (this corresponds to the REC price in a real market). A sensitivity analysis can find the REC price that just complies with the RPS constraint. It follows that, for increasing RPS requirements, the REC price will increase to incentivize compliance. Table 5.3 summarizes the findings of this section.

Table 5.3: Renewable Portfolio Standard Compliance and Renewable Utilization

	Expected RPS Compliance		Expected Renewable CF	
	No ESS	With ESS	No ESS	With ESS
0% RPS / $0 REC	7%	14%	33%	32%
25% RPS / $25 REC	20%	43%	32%	31%
40% RPS / $50 REC	34%	57%	34%	31%
75% RPS / $125 REC*	45%	76%	30%	27%

From Table 5.3, we can see that RPS compliance nearly doubles when ESS is available in each RPS scenario; however, the expected utilization of renewable resources–defined here as the capacity factor for all renewables–does not change significantly. As such, we gather that

ESS is not helping better capture renewable resources but simply making over-investment in renewable capacity economically viable for the system. This argument will be verified in the following subsections. While four RPS scenarios were tested, we present only two extreme cases in this paper: 0% and 75%. It should be noted that the 75% RPS scenario with ESS was only solved to 1.5% optimality gap due to time limitations.

5.3.1. 0% RPS with $0 REC. Initially, we test the model with no RPS requirement as a base case. Comparing, Figure 5.1 to Figure 5.2, we see significant changes in operational behavior in the system. This is most evident in the flattening of thermal generation and almost 70% reduction in peak unserved energy (load shed). With ESS, 14% of the total generation comes from renewables, compared to just 7% in the scenario without ESS, showing that ESS is an economic choice for increasing the value of renewables in the system.

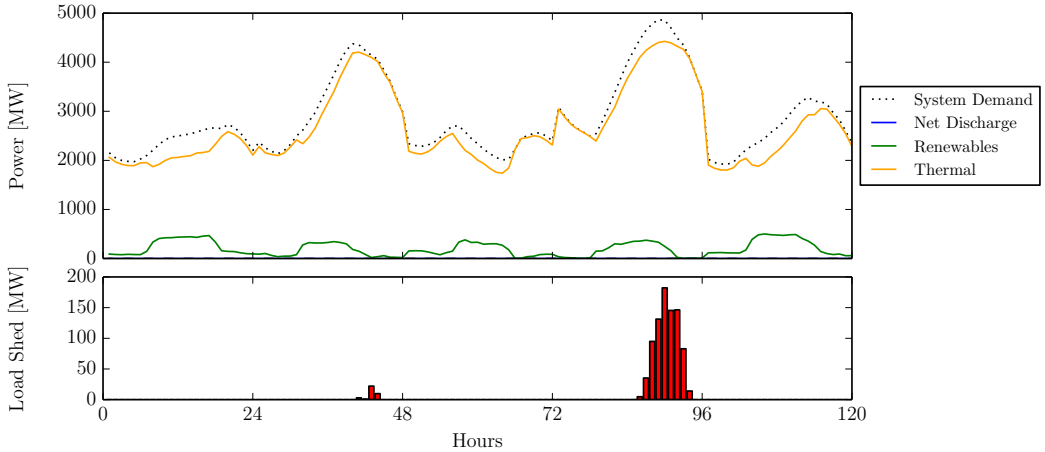


Fig. 5.1: Operational results without energy storage for five sampled days given the default capital costs associated with ESS ($50/kWh and $1000/kW over a 20 year operational life). Note y-axis scales change between figures.

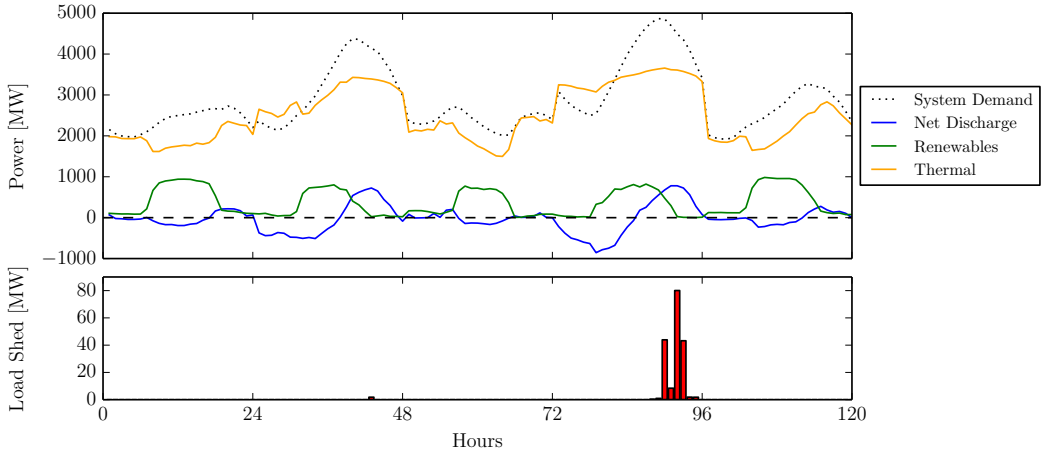


Fig. 5.2: Operational results with energy storage for five sampled days given the default capital costs associated with ESS ($50/kWh and $1000/kW over a 20 year operational life). Note y-axis scales change between figures.

5.3.2. 75% RPS with $125 REC*. Over the long term, we can expect that RPS requirements could be pushed to even higher levels, such as 75%. To test such a scenario, we again run the model again with a 75% RPS requirement and a $125 REC.

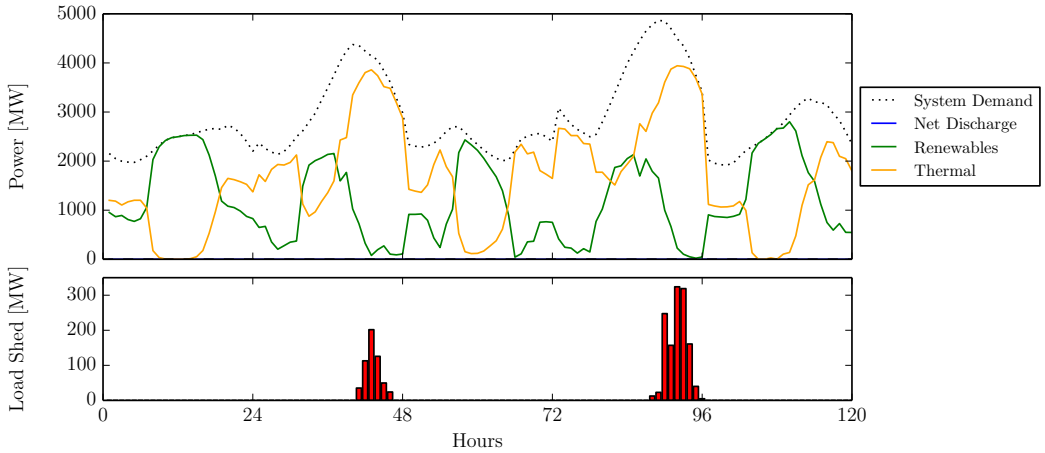


Fig. 5.3: Operational results without energy storage for five sampled days given a 75% RPS requirement and associated $125 Renewable Energy Certificate price. Note y-axis scales change between figures.

In Figure 5.3, the model invests just enough capacity to meet the lowest mid-day system demand, so that peak renewable generation never exceeds system demand. This is a rational choice for the model, since energy (demand or supply) cannot be shifted to other times of the day. Because of this limitation, the model only achieves a 45% RPS compliance.

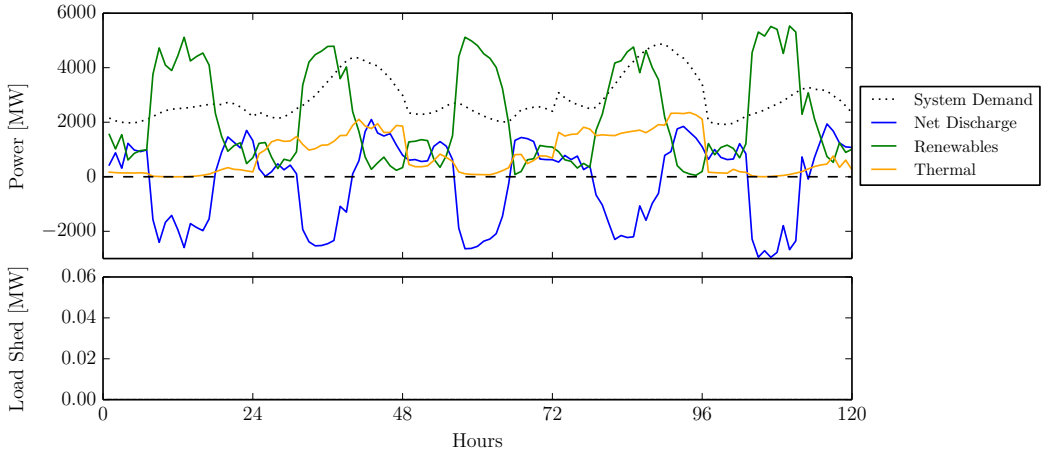


Fig. 5.4: Operational results with energy storage for five sampled days given a 75% RPS requirement and associated $125 Renewable Energy Certificate price. Note y-axis scales change between figures.

In contrast, Figure 5.4 shows that the model with ESS available significantly over-invests in renewable generation capacity such that peak renewable generation meets or exceeds system demand for every sampled day. ESS captures any excess renewable generation and discharges throughout off-peak periods, providing cost savings in the form of reduced thermal dispatch. As in the 0% RPS requirement scenario, we continue to see both significant flattening of the thermal generation time series and reductions in unserved energy.

5.4. ESS Operational Behaviors. We observe two types of operational behavior from the operational time series plots: 1) arbitrage and 2) peak shaving. In low-peak days, ESS charges significantly when cheap, renewable (solar PV) production is highest during the day and discharges at all other times, when costly thermal generators would otherwise be necessary. While our model does not explicitly consider unit commitment and ramping decisions, it is reasonable to expect that adding these constraints would further favor this kind of arbitrage behavior. On the high-peak days, we see the second behavior, where the storage switches from net-charge to net-discharge around midday to help the system meet peak demand. Again, if unit commitment and ramping decisions were considered in the system, this peak shaving behavior could produce further net cost savings.

6. Conclusions and Future Work. Using a stochastic transmission expansion planning formulation based on economic dispatch, we are able to optimize bulk energy management using ESS in conjunction with generation and transmission expansion. In small test cases of only five sampled days, we see that ESS is able to add value to the system by reducing the likelihood of unserved energy in sampled days and, more importantly, increasing RPS compliance.

There are significant computational hurdles to solving the extensive form of this mixed-integer expansion planning model for systems of realistic size or with many sampled days, both of which would be required for practical use, that are beyond the scope of this paper. As such, future work based on [19] will implement a novel Benders decomposition algorithm to allow us to parallelize and solve the problem on a high performance computing platform in a reasonable amount of time. We will use a 240-bus representation of the Western Energy

Coordinating Council (WECC) system as a case study for this algorithm.

REFERENCES

[1] M. Abdurrahman, S. Baker, B. Keshavamurthy, and M. Jacobs, *Energy storage as a transmission asset*, tech. rep., PJM Interconnection, 2012.

[2] D. Azari, S. S. Torbaghan, M. Gibescu, and M. A. van der Meijden, *The Impact of Energy Storage on Long Term Transmission Planning in the North Sea Region*, in IEEE Power and Energy Society General Meeting, 2014.

[3] Bagen and R. Billinton, *Impacts of Energy Storage on Power System Reliability Performance*, in Canadian Conference on Electircal and Computer Engineering, 2005, pp. 494–497.

[4] R. Christie and C. Grigg, *University of Washington Power Systems Test Case Archive, IEEE RTS 1996*.

[5] J. Cochran, M. Miller, M. Milligan, E. Ela, D. Arent, and A. Bloom, *Market Evolution: Wholesale Electricity Market Design for 21st Century Power Systems*, tech. rep., National Renewable Energy Laboratory, October 2013.

[6] A. J. Conejo, M. Carrión, and J. M. Morales, *Decision Making Under Uncertainty in Electricity Markets*, Springer Science+Business Media, 2010.

[7] EnerNex Corporation, *Eastern Wind Integration and Transmission Study*, Tech. Rep. NREL/SR-5500-47078, National Renewable Energy Laboratory, 2011.

[8] A. Faghih, M. Roozbehani, and M. A. Dahleh, *Optimal Utilization of Storage and the Induced Price Elasticity of Demand in the Presence of Ramp Constraints*, in IEEE Conference on Decision and Control and European Control Conference, 2011.

[9] M. Hadayati, J. Zhang, and K. W. Hedman, *Joint Transmission Expansion Planning and Energy Storage Placement in Smart Grid Towards Efficient Integration of Renewable Energy*, in IEEE Power and Energy Society T&D Conference and Exposition, 2014.

[10] W. E. Hart, C. Laird, J.-P. Watson, and D. L. Woodruff, *Pyomo–Optimization Modeling in Python*, vol. 67, Springer Science+Business Media, 2012.

[11] Z. Hu, F. Zhang, and B. Li, *Transmission Expansion Planning Considering the Deployment of Energy Storage Systems*, in IEEE Power and Energy Society General Meeting, 2012.

[12] B. Kaun, *Cost-Effectiveness of Energy Storage in California: Application of the EPRI Energy Storage Valuation Tool to Inform the California Public Utilities Commission Proceeding R. 10-12-007*, Tech. Rep. 3002001162, Electric Power Research Institute, 2013.

[13] M. E. Khodayar, M. Shahidehpour, and L. Wu, *Enhancing the Dispatchability of Variable Wind Generation by Coordination with Pumped Storage Hydro Units in Stochastic Power Systems*, IEEE Transactions on Power Systems, 28 (2013), pp. 2808–2818.

[14] M. Koller, T. Borsche, A. Ulgib, and G. Andersson, *Defining a Degradation Cost Function for Optimal Control of a Battery Energy Storage System*, in PowerTech, IEEE Grenoble, 2013, pp. 1–6.

[15] I. Konstantelos and G. Strbac, *Valuation of Flexible Transmission Investment Options Under Uncertainty*, IEEE Transactions on Power Systems, 30 (2015), pp. 1047–1055.

[16] N. Li and K. W. Hedman, *Economic Assessment of Energy Storage in Systems with High Levels of Renewable Resources*, IEEE Transactions on Sustainable Energy, 6 (2015), pp. 1103–1111.

[17] T. Mai, R. Wiser, D. Sandor, G. Brinkman, G. Heath, P. Denholm, D. J. Hostick, N. Darghouth, A. Schlosser, and K. Strezepek, *Exploration of High-Penetration Renewable Electricity Futures. Vol. 1 of Renewable Electricity Futures Study*, Tech. Rep. NREL/TP-6A20-52409-1, National Renewable Energy Laboratory, 201.

[18] N. Miller, M. Shao, S. Pajic, and R. D'Aquila, *Western Wind and Solar Integration Study Phase 3 – Frequency Response and Transient Stability*, Tech. Rep. NREL/SR-5D00-62906, GE Energy Management, National Renewable Energy Laboratory, 2014.

[19] F. D. Muñoz, *Engineering-Economic Methods for Power Transmission Planning under Uncertainty and Renewable Resource Policies*, PhD thesis, Johns Hopkins University, January 2014.

[20] F. D. Muñoz and A. D. Mills, *Endogenous Assessment of the Capacity Value of Solar PV in Generation Investment Planning Studies*, tech. rep., Sandia National Laboratories and Lawrence Berkeley National Laboratories, January 2015.

[21] F. D. Muñoz and J.-P. Watson, *A Scalable Solution Framework for Stochastic Transmission and Generation Planning Problems*, Computational Management Science, (2015).

[22] B. Palmintier, *Flexibility in Generation Planning: Identifying Key Operating Constraints*, in Power Systems Computation Conference, August 2014.

[23] H. Pandžić, Y. Wang, T. Qiu, Y. Dvorkin, and D. S. Kirschen, *Near-Optimal Method for Siting and Sizing of Distributed Storage in a Transmission Network*, IEEE Transactions on Power

Systems, 30 (2015), pp. 2288–2300.

[24] I. J. Pérez-Arriaga, *Managing Large Scale Penetration of Intermittent Renewables*, tech. rep., Massachusetts Institute of Technology Energy Institute, 2011.

[25] PJM Interconnection, *Transmission System Operations TO1, Interconnection Training Program.* Presentation, 2011.

[26] U.S. Energy Information Agency, *Updated Capital Cost Estimates for Utility Scale Electricity Generating Plants*, tech. rep., April 2013.

[27] ———, *Short Term Energy Outlook*, tech. rep., June 2015.

[28] S. Vajjhala, A. Paul, R. Sweeney, and K. Palmer, *Green Corridors: Linking Interregional Transmission Expansion and Renewable Energy Policies*, tech. rep., Resources for the Future, 2008.

[29] V. Viswanathan, M. Kintner-Meyer, P. Balducci, and C. Jin, *National Assessment of Energy Storage for Grid Balancing and Arbitrage, Phase II, Volume 2: Cost and Performance Characterization*, Tech. Rep. PNNL-21388, Pacific Northwest National Laboratory, September 2013.

[30] J.-P. Watson, D. L. Woodruff, and W. E. Hart, *PySP: Modeling and Solving Stochastic Programs in Python*, Mathematical Programming Computation, 4 (2012), pp. 109–149.

[31] Y. Wen, G. Chuangxin, and S. Dong, *Coordinated Control of Distributed and Bulk Energy Storage for Alleviation of Post-Contingency Overloads*, Energies, 7 (2014), pp. 1599–1620.

EFFICIENT DESTINATION PREDICTION USING AIRCRAFT TRAJECTORY DATA

BENJAMIN D. NEWTON*, MARK D. RINTOUL†, CHRIS G. VALICKA‡, AND ANDREW T. WILSON§

Abstract. Efficiently and accurately predicting an expected future destination, given only the past trajectory of some object, person, or vehicle is desirable in many application domains. In this work we outline a novel method for predicting destinations given a data set of historical trajectories. The method works by first creating feature-vectors for multiple partial trajectories for each historical trajectory, then finding, in this high-dimensional space, nearest neighbors to a feature vector created for a new trajectory. Our method is tested on a large data set of real-world aircraft trajectories. We show examples of where our method works well, and where it could be improved, and demonstrate that with a large training data set destination airports can be predicted with 63% accuracy, and 85% of destination airports can be predicted correctly given 5 guesses, using just the first half of an unfinished trajectory.

1. Introduction. Within the deluge of data being analyzed by today's data scientists are many instances of time-indexed position information, called trajectories. The analysis of trajectories has been well studied in many fields, and efforts continue to improve trajectory analysis techniques [7]. In this work we concentrate on the objective of efficiently predicting the eventual destination location of an unfinished trajectory given a database of historical trajectories. We make no assumption about potential constraints on the routes, such as a network of roads, and we desire to be able to quickly obtain results given a data set of millions or even hundreds of millions of trajectories. Predicting destinations is applicable to many problem domains. For example, with accurate destination prediction a user's smart phone could determine where the user is likely traveling, and inform them of their estimated time of arrival and/or alert the user of potential issues (i.e. dynamic traffic alerts) or opportunities along the route. In addition, if a hybrid vehicle were able to accurately predict its expected route and destination, its energy usage could be optimized by, for example, delaying a battery recharge until regenerative braking could be used on an upcoming hill [2]. One could even imagine trajectories being extracted eye-tracking data (time-indexed positions of where a user is looking on a screen), and predicting in real-time to which position on a screen a user's eyes were moving.

Our approach exploits two characteristics exhibited by many types of trajectory data. First, a very similar route is generally taken from a specific origin to a specific destination. Second, most new trajectories are very similar to a previously seen trajectory, given enough historical data [3]. Given these facts, to predict an expected destination we first generate multiple feature vectors in a high-dimensional space for each historical trajectory. Then, nearest neighbors are found, in the feature-space, for the unfinished trajectory whose destination we wish to predict. Confidence values are assigned to each of the most similar trajectories, and finally the probable destination location is determined.

We use a large database of aircraft trajectory data to demonstrate the efficacy and efficiency of this method for predicting trajectory destination locations. Although our method is general enough to work on other types of trajectories, analyzing air traffic allows us to experiment with a large-scale real-world database, where origin and destination locations are generally well-defined.

*The University of North Carolina at Chapel Hill, bn@cs.unc.edu
†Sandia National Laboratories, mdrinto@sandia.gov
‡Sandia National Laboratories, cgvalic@sandia.gov
§Sandia National Laboratories, atwilso@sandia.gov

The remainder of this paper is organized as follows. In the next section we summarize several works related to trajectory destination prediction. Next, in Section 3 the problem we desire to solve is formally defined, and section 4 describes our trajectory data set. Section 5, then, details each stage of our method, and in Section 6 various results of predicting destinations with our method are discussed. Finally in Section 7 we describe some opportunities for future work, and conclude.

2. Related Work. Trajectory destination prediction has been studied in several different contexts, and using several different methods. Some of the most relevant research is summarized below.

Froehlich and Krumm describe their route prediction system in [3]. They use a version of the Hausdroff distance to compare and Dendogram clustering to merge cleaned trip trajectories into common routes. Using similar methods, partial trajectories were then compared with the routes to obtain a predicted destination. The method was tested on a database of automobile trajectories collected from 250 drivers. The results show that half way through a trip, the future route can be correctly predicted 20% of the time, and the correct route is in the top 10 matches 40% of the time. Considering only repeat trips, the same values jump to 40% and 97% respectively. Because this method must compare every pair of trajectories in the input (n^2 comparisons), it will likely not scale well to large data sets, such as the hundreds of thousands of aircraft trajectories we desire to analyze.

In [5] Krumm and Horvitz detail Predestination, a system for inferring destinations from partial trajectories. They utilize Bayesian inference to produce a probabilistic map of potential destinations, which results in a median error of 3 to 5 km in predicting the destination at the half-way point of a trip. Patterson et al. [6] applied machine learning and a particle filter to GPS traces to predict a person's destination, future route, and even mode of transportation. This method is able to make good predictions for a few blocks into the future in an urban environment, but unfortunately does not make accurate long-term predictions. Simmons et al. [10] rely on information about a road network, in their task of predicting driver routes and destinations. They generate a Hidden Markov Model (HMM) of the routes and destinations visited by a driver, and then use the model to predict future behavior. Our desire, however, is a more general method which does not rely on information about an underlying network of roads or airways.

Researchers at Motorola [11] have also developed a system for learning routes from a users travel data. Learned routes are stored in a database which can then be queried to predict the destination of a current route. Based upon the predicted destination and future route, appropriate traffic advisories can be sent to the user.

Chen et al. [1] developed a system for predicting future routes and destinations, but rather than focus on vehicular trajectories, they collected real personal movement data from a small set of participants, and mined this data to enable accurate predictions. Movement data was first mined to extract a set of significant places from which destination and origin locations could be predicted. Next, the system extracted abstract movement patterns from the training data. Finally, in a separate module the movement patterns were used to construct a pattern tree, that is traversed to find matches to the live movement data. One focus of their work was to predict both the future route and the destination in a unified manner, unlike previous approaches for which the predicted route may not match well with the predicted destination. The researchers showed their method could predict the destination of a user with 79.6% accuracy. Our method, however, is more general and does not seek to handle or utilize the unique qualities of personal movement data.

3. Problem Definition. The problem we desire to solve is defined as follows. Given a large-scale data set of historical trajectories, and without explicitly assuming anything

about the geographical limits placed on those trajectories (such as a road network), how can the eventual destination of a new unfinished trajectory be accurately and efficiently predicted? One key aspect of our problem is the scale. We desire to be able to train using a database of at least 1 million trajectories each made up of 100 points, on average. Our training should be efficient, and not require direct comparisons between every trajectory. Training should only require hours, and predicting a single destination should take a fraction of a second. Finally the prediction results should be relatively accurate.

4. Aircraft Trajectory Data. The United States FAA (Federal Aviation Administration) makes available, to certain industry and research partners, position information for most active flights in the National Airspace System (NAS). This data, known as Aircraft Situation Display to Industry (ASDI), includes time-stamped position and velocity information for each aircraft, and often other relevant values such as origin airport code, destination airport code, and flight plan information. We have collected a large set of ASDI data which we use to analyze our destination prediction method.

The raw ASDI data consists of millions of position reports (about 5 million for each day of data in 2015). Each of these reports includes a flight call sign (such as DAL1835), but there is no organization or division of the ASDI data into individual flights.

5. Our Approach. Our method predicts destinations by advancing through a series of stages, including data cleaning, feature vector creation, R-tree building, similar trajectory finding, and analysis. Each of these stages are described in detail below.

5.1. Cleaning The Data. As described in Section 4, the ASDI data is not organized or divided into individual flights. We first group the ASDI data into position reports with the same call sign, but further refinement is necessary, because the same call sign may be used for a similar flight each day, or even different flights during the same day. A heuristic is used to separate position reports for a given call sign into individual flights. The separation criteria consists of three components: the maximum separation time between consecutive position reports, the maximum separation distance between consecutive position reports (unused for the analysis below), and the minimum number of position reports for a flight. A time-sorted list of position reports for a given call sign is split into separate flights where a time gap larger than the maximum separation time, or a distance gap larger than the maximum separation distance is found. Further, if this splitting yields a set of position reports whose size is less than the given minimum number, the set of position reports is discarded. The result is about 45,000 separate trajectories for each day of ASDI data (in 2015).

Our data is now separated into a set of flight trajectories, with each flight trajectory containing a set of position reports. Because of the noisiness of the data, we perform another filtering step, using a heuristic to identify suspicious position reports in each flight trajectory. A position report is considered suspicious if, (1) it occurs abnormally soon after a previous position report, (2) its reported position is far away from the previously reported position, (3) its reported altitude is far above or below the previously reported altitude, (4) its reported altitude is below some threshold value (for example 0). All suspicious position reports are removed from the flight trajectories.

In a final filtering step, we also discard any flight which now contains fewer than the minimum number of position reports for a flight, any flight whose call sign indicates that it is a general aviation flight, and any flight whose origin airport or intended destination airport is invalid. Each position report includes the origin airport and the intended destination airport. An airport is considered invalid if, (1) it is not reported, (2) it changes between the first and last position reports, or (3) the actual trajectory start or end is not within 50

km of the reported origin or intended destination. Removing flights with invalid airports allows us to have unambiguous ground-truth data. There are many trajectories in our data set, for example, which start or terminate abruptly due to technical difficulties, or at the edge of the tracked space. Including these trajectories, which do not terminate at the intended airports, yields degraded results. Similarly, removing general aviation traffic allows us to concentrate on commercial and cargo air traffic. The result of this pre-processing is a database of trajectories (about 11,000 for each day of 2015 data) which we can now use to assist in destination prediction.

The output of this step is a set of trajectories where each trajectory is comprised of $n+1$ time-stamped points $(x_0, t_0), (x_1, t_1), ..., (x_n, t_n)$, where x_i is the position of point i. In our example application of predicting aircraft destinations, the positions are latitude, longitude pairs, however the our method also works for other coordinate systems.

5.2. Creating Feature Vectors. We desire to a way to succinctly summarize a trajectory (or fraction of a trajectory). Combining a small set of normalized trajectory characteristics into an N-dimensional feature vector will enable us to find similar trajectories in and N-dimensional space, using traditional techniques.

Two issues must be overcome for this approach to work. First, an appropriate set of trajectory characteristics to use in feature vector creation must be determined. Second, since our goal is to predict the destinations of flights for which only a fraction of the trajectory is given, the feature vectors created must allow for matching partial trajectories.

A wide variety of characteristics have been used to summarize trajectories in trajectory analysis [7] applications such as finding similarly shaped trajectories, or identifying anomalous trajectories. Often, characteristics for these applications are chosen such that they are translation invariant. For the application of finding destinations, however, translation invariance is not desirable. Instead, trajectories which basically follow the same path, without translation, must be found. To accomplish this, we choose a set of N sample points equally spaced in time along the trajectory. The positions are determined by linear interpolation (along the great circle path), when they are not aligned with an existing position report. Each sample point contributes to the feature vector two values, the latitude and longitude components of the position. These sample points provide a succinct representation of the trajectory. The feature vector may also include additional values representative of the entire trajectory, such as total flight duration. As an example, setting N to 4, and using a single additional value of duration, yields a 9-dimensional feature vector.

To be able to match partial trajectories we simply create feature vectors for progressively larger portions of each original trajectory. For example, feature vectors could be generated for the first 1/4, the first half, and the first 3/4 of an original trajectory, generating 3 feature vectors from one trajectory. Creating feature vectors in this manner, enables better destination prediction for trajectories of in-progress flights. Users may select a lower bound, an upper bound, and a number of feature vectors per input trajectory as parameters to control how feature vectors are generated. For even better results when large training data sets are utilized, we select random-length portions of the original trajectory.

5.3. Building an R-tree. Next, the generated feature vectors are stored in a data structure known as an R-tree [4]. An R-tree is a data structure which enables efficient searching for nearest neighbors in multi-dimensional data. Our implementation builds upon boost's implementation of an R-tree, adding specialized find methods. For this stage of the prediction process, we simply insert the feature vectors created for the input training data set of trajectories into an instance of the R-tree data structure.

5.4. Finding Similar Trajectories. Now, given a new trajectory for which we desire to predict a destination, we need only create a feature vector the trajectory (using the same method described above), and find its K nearest neighbors in the R-tree. For each similar trajectory i and j, a confidence value c_{ij} is generated to measures how well the trajectories match. The confidence value is essentially the inverse of the distance between the points in feature-space squared, and is computed using the following formula:

$$c_{ij} = \frac{1.0}{0.01 + {d_{ij}}^2} \tag{5.1}$$

where d_{ij} is the distance between the feature vectors i and j.

5.5. Performing Analysis. The similar trajectories and associated confidence values can now be analyzed to predict destinations. We describe two different methods of analysis depending on the desired output, though many others could be utilized. For the first method, we associate each trajectory in the data set with a destination airport code (i.e. LAX, for Los Angeles International Airport). The confidence values of those trajectories (from the set of similar trajectories found) with matching destination airport codes are summed. The result is a weighted list of potential destination airport codes. From this list the trajectory with the highest confidence can be reported as the expected destination. If multiple guesses are allowed, we can proceed down the list until a threshold weight or number of guesses are reached.

For applications where a set of potential destination locations (such as airports) is not available, the actual expected destination position can be computed. This is accomplished by taking the weighted spherical linear interpolation (slerp) [9] of the destination positions of the similar trajectories found, using their associated confidence values as weights. The result will be a point which is the weighted "geographic" mean of the candidate trajectory destinations.

6. Results. We tested our destination prediction method with the Tracktable Trajectory Analysis library [8]. Tracktable is an open source library which contains a core set of functionality for ingesting, processing, plotting, and analyzing trajectories. We have subsequently incorporated or prediction method into the library. This section describes the parameters used, and the results obtained using our method to predict destinations of real aircraft trajectories.

6.1. Parameters. Our data cleaning and destination prediction rely on several parameters. Table 6.1 lists these parameters, and the values used to obtain the results described below. The first 5 parameters are used to filter out bad points and bad trajectories. The next four parameters adjust how feature vectors are created for a trajectory, and how multiple feature vectors are generated for different fractions of an input trajectory. For this analysis we use 4 position samples per trajectory, and an additional value, representing the duration of the partial trajectory. Finally, the last parameter is used to determine the number of nearest neighbors to utilize while finding similar trajectories.

6.2. Matching Trajectories. First, we show an example of our method successfully finding several similar flights with a common destination. We select as a test trajectory the path taken on an inter-island flight in Hawaii from Kona to Honolulu. Using only features from the first half of this trajectory, and the duration of the first half of the flight, we employ our method to find matches and predict the destination given a training set of only 8,636 trajectories spread across the United States. In Figure 6.1 the trajectories of the test trajectory and 9 matching trajectories are plotted on a map of the Hawaiian Islands. As with other figures in this work, the trajectories are plotted with lines whose color slowly

Trajectory Splitting and Cleaning	min trajectory length	20 samples
	max separation time between consecutive points	10 minutes
	min separation time between consecutive points	30 seconds
	max distance between consecutive points	60 nautical miles
	max altitude change between consecutive points	75,000 feet
	min altitude	0 feet
	max distance between airport and trajectory end	50 km
Feature Vector Creation	N (samples per trajectory)	4
	additional value in each feature vector	duration
	low	0.1, 0.2 for table data
	high	1.0, 0.8 for table data
	feature vectors per trajectory	10, 7 for table data
Prediction	K (num nearest neighbors)	10

Table 6.1: Parameter values for cleaning data and predicting destinations

transitions from red, at the origin, to blue, at the destination. The matching trajectories all depart from the Kona International Airport (KOA) and arrive at the Honolulu International Airport (HNL), giving us a very high confidence that the intended destination of this flight is the Honolulu airport. While this is a straight-forward example, since there are few other likely destinations along this path in the vast ocean, it allows us to examine some aspects of the method. First, notice that our method finds trajectories which follow the same general path but can exhibit some variability. Also note that because our test trajectory is a partial trajectory, utilizing only feature points along the first half of its total path, there is generally more variation in the trajectories after the midway point. One flight, in particular juts out to the north east just after the midway point. This trajectory would not have matched as strongly had the test trajectory been slightly longer. Figure 6.2 similarly shows a successful prediction for a flight from Seattle Washington to Portland Oregon. Notice how perfectly the trajectories in the first half of the flight are aligned.

6.3. Mismatched Trajectories. As could be expected, matching a test trajectory is not always so easy. Figure 6.3 shows a region of the western United States, with Wyoming in the center, Utah on the left, and South Dakota in the upper right. Plotted are a test trajectory for Sky West flight 125F departing Rapid City Regional Airport (RAP) in South Dakota, and arriving at Salt Lake International Airport (SLC) in Utah, and the top 5 similar trajectories found using our method when using a training data set containing 21,569 trajectories of flights across the United States all on the same day, July 10, 2013. As you can expect, the destination prediction for this case is a dismal failure, as none of the matching trajectories even share the same destination. Notice that there are even matching trajectories that fly the opposite direction of the test trajectory, from south to north. The

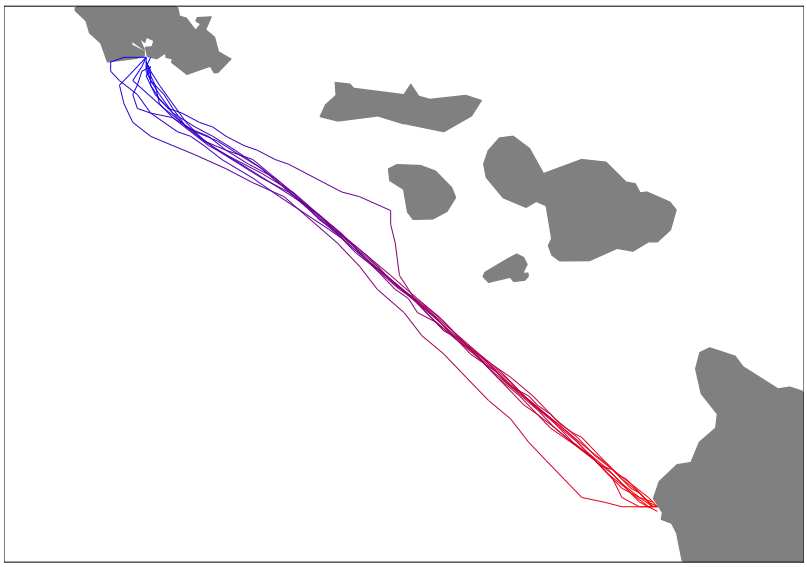


Fig. 6.1: An example of 9 flights used to accurately predict the destination of a test trajectory, the first half of a flight from Kona to Honolulu Hawaii. (red=origin, blue=destination)

failure of our method in this case is likely due to insufficient training data. This appears to be an uncommon route, perhaps flown only once a day, or less. If predictions are to be accurately made on uncommon flights, such as this, our method requires more training data, perhaps weeks or months of data.

6.4. Prediction Accuracy. We evaluate the accuracy of our method using one month of historical trajectory data from July 2015. Cleaning the large data set results in 323,605 trajectories. We employ a leave-one-out approach for this analysis. One trajectory is left out of the training data set, and then the destination for a random fraction of that trajectory is predicted. This is repeated for each input trajectory, such that each is "left out" in turn. By a random fraction of a trajectory we mean a partial trajectory starting at the origin and extending to a point, a random temporal fraction of the way towards the destination. Using only a fraction of the trajectory enables realistic testing of predicting destinations of unfinished trajectories. The random fraction for this analysis is selected anywhere within the same lower and upper bounds used for creating feature vectors (see Table 6.1 and note we use 0.2 and 0.8 for the lower and upper values here). We find the 10 nearest neighbors, and extract the destination airport codes from those matches. We then merge results for identical airport codes, to obtain the top potential matches. Table 6.2 shows the results of this analysis.

The results indicate that our method is able to obtain relatively good accuracy given a large training data set. While for about 14% of the flights the destination was not able to be predicted, over 64% of our top matches were correct. An additional 12% (76.7%) of the second matches were correct, and nearly 86% of the destination were able to be correctly predicted given up to 5 guesses.

A more detailed analysis is obtained by generating feature vectors for 10 random portions of each trajectory in the data set, where each random portion extends from the origin

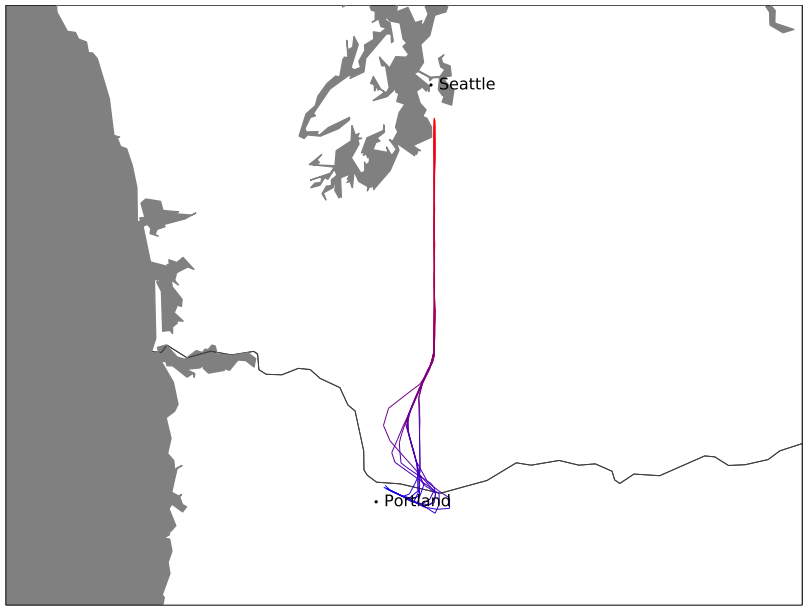


Fig. 6.2: An example of 10 flights used to accurately predict the destination of a test trajectory, the first half of a flight from Seattle, Washington to Portland, Oregon. (red=origin, blue=destination)

	Number	Accuracy
Not Matched	44,390	13.7%
Top Match	208,439	64.4%
Within Top 2 Matches	248,316	76.7%
Within Top 5 Matches	276,756	85.5%
Within Top 10 Matches	279,215	86.2%

Table 6.2: Accuracy given 1 months worth of historical aircraft trajectories, for a first random fraction for each left out trajectory.

to 10 to 100% of the way to the destination (low=0.1, high=1.0). We again employing a leave-one-out analysis, and predict the destinations of 10 random portions of each trajectory also extending from the origin to 10 to 100% of the way towards the destination. Figure 6.4 shows the results of this analysis where each data point is the number of correct predictions for all the random samples of a given percentage of flight length, for various numbers of top matches.

The figure shows that half way through a flight the correct destination can be predicted as the top match 63% of the time, and in the top 5 matches 85% of the time. There is also little difference between using the top 5 and top 10 matches, suggesting that fewer than 10 matches could be computed, with minimal impact. The rise at the beginning of the plot is likely a boundary effect, since there are no feature vectors generated for the first 10% of the flights.

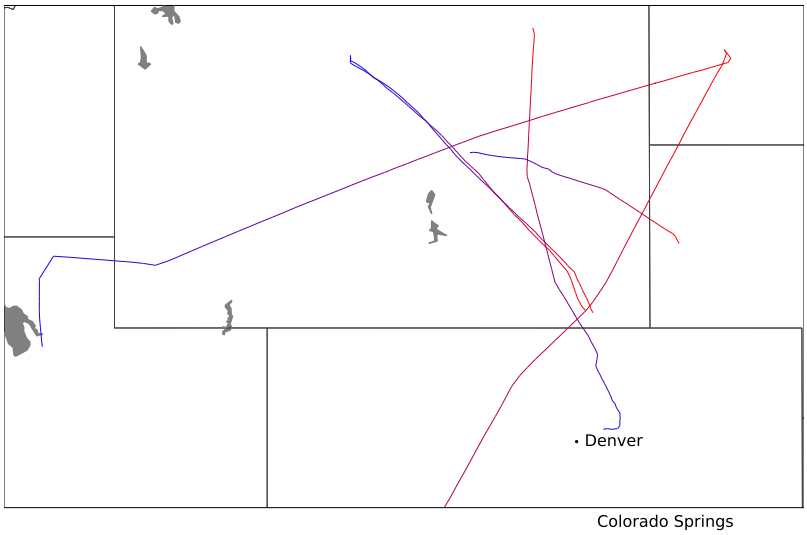


Fig. 6.3: An example where our prediction method fails. A test trajectory from Rapid City to Salt Lake City is plotted, as well as the top 5 matching, which show little similarity due to a lack of training data. (red=origin, blue=destination)

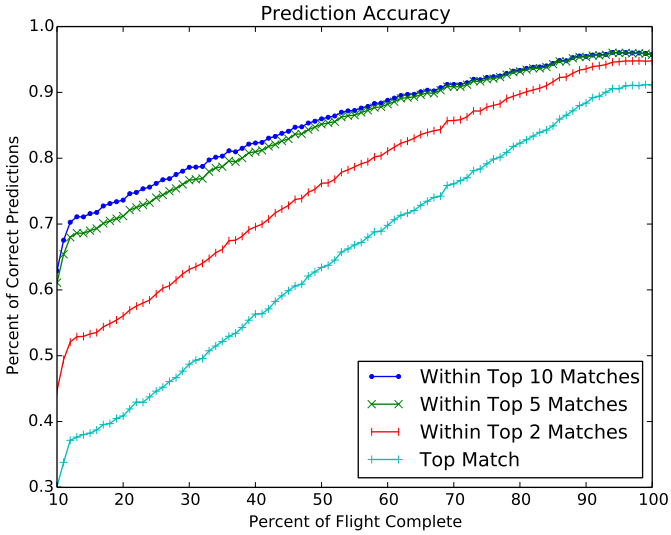


Fig. 6.4: Accuracy obtained using various numbers of top matches, employing the leave-one-out method.

7. Future Work and Conclusion. We have described our novel method for predicting destinations of incomplete trajectories by finding similar trajectories in a a high-dimensional space populated with feature vectors derived from historical trajectories. Our method is efficient, in that it does not need to directly compare the new trajectory to all historical trajectories, and it is relatively accurate, given a large data set.

Several aspects of our method could be improved with further effort. Accuracy could likely be improved by utilizing the Hausdroff Distance Algorithm or other similarity algorithms to further refine predictions once a small set of candidates are identified. Also, our method is highly parallelizable, and we are working towards efficiently using multiple processors to speed-up the processing stages, and R-tree look-up.

Data scientists deal with large data sets of trajectories, where similar trajectories are often traveled from a set of common origin and destination points, they now have a new tool to efficiently predict probable destinations of unfinished trajectories. We hope that this new method will enable further progress in analyzing and predicting behaviors in various data sets.

REFERENCES

[1] L. Chen, M. Lv, and G. Chen, *A system for destination and future route prediction based on trajectory mining*, Pervasive Mob. Comput., 6 (2010), pp. 657–676.

[2] Y. Deguchi, K. Kuroda, M. Shouji, and T. Kawabe, *Hev charge/discharge control system based on navigation information*, in SAE Convergence International Congress and Exposition On Transportation Electronics, Detroit, Michigan USA, 2004.

[3] J. Froehlich and J. Krumm, *Route prediction from trip observations*, in Society of Automotive Engineers (SAE) 2008 World Congress, 2008.

[4] A. Guttman, *R-trees: A dynamic index structure for spatial searching*, SIGMOD Rec., 14 (1984), pp. 47–57.

[5] J. Krumm and E. Horvitz, *Predestination: Inferring destinations from partial trajectories*, in Proceedings of the 8th International Conference on Ubiquitous Computing, UbiComp'06, Berlin, Heidelberg, 2006, Springer-Verlag, pp. 243–260.

[6] D. Patterson, L. Liao, D. Fox, and H. Kautz, *Inferring high-level behavior from low-level sensors*, in UbiComp 2003: Ubiquitous Computing, A. Dey, A. Schmidt, and J. McCarthy, eds., vol. 2864 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2003, pp. 73–89.

[7] M. D. Rintoul and A. T. Wilson, *Trajectory analysis via a geometric feature space approach*, to appear in Statistical Analysis and Data Mining, (2016).

[8] Sandia National Labs, *Tracktable*, 2015. http://tracktable.sandia.gov.

[9] K. Shoemake, *Animating rotation with quaternion curves*, SIGGRAPH Comput. Graph., 19 (1985), pp. 245–254.

[10] R. Simmons, B. Browning, Y. Zhang, and V. Sadekar, *Learning to predict driver route and destination intent*, in Intelligent Transportation Systems Conference, 2006. ITSC '06. IEEE, Sept 2006, pp. 127–132.

[11] K. Torkkola, K. Zhang, H. Li, H. Zhang, C. Schreiner, and M. Gardner, *Traffic advisories based on route prediction*, in Workshop on Mobile Interaction with the Real World (MIRW 2007), Singapore, 2007.

UNCERTAINTY QUANTIFICATION OF THE INTERFACIAL MASS TRANSFER MODEL IN CTF

NATHAN W. PORTER* AND VINCENT A. MOUSSEAU†

Abstract. This work analyzes the uncertainty of the bulk mass transfer model in the legacy thermal hydraulic subchannel code Coolant Boiling in Rod Arrays - Three Field (COBRA-TF). The authors' contribution to the existing code documentation is presented, with focus on the inherent difficulty of working with legacy codes - particularly when performing verification, validation and uncertainty quantification (VVUQ). Black box uncertainty quantification methods are very limiting because they ignore many numerical parameters that may contribute significantly to the simulation uncertainty. Uncertainty quantification of these numerical parameters (flow regime map, spatial smoothing, under relaxation, etc.) will require future work. An example of holistic uncertainty analysis is demonstrated for a single correlation using Bayesian calibration. This method allows the experimental data from many sources to directly inform correlation uncertainty, which can be directly incorporated into a simulation tool.

1. Introduction. COBRA-TF is a thermal hydraulic subchannel code that uses eight conservation equations to solve models of light water reactor (LWR) in-core geometries. The entrained liquid is assumed to be at thermal equilibrium with the continuous liquid, so there is no energy equation for the droplet phase. The code has extensive models for a variety of thermal hydraulic phenomena and has been used throughout industry and academia to model a large number of reactor problems. This code was originally developed as part of COBRA/TRAC at Pacific Northwest National Laboratory in 1980 [15]. CTF is one of the many versions that have evolved from the original and is developed by Pennsylvania State University (PSU) and the Consortium for Advanced Simulation of LWRs (CASL) [14].

Codes written the 1970's and 1980's, like CTF, are often referred to as "legacy" codes in the nuclear industry. These were designed decades ago when computational limitations required a large number of simplifications. It is well known that these simplifications are not generally applicable and that they introduce significant amounts of uncertainty. Creation of a new code is an expensive and time-consuming endeavor; modernizing a legacy code is often preferred, but this process presents a variety of unique challenges.

These codes were originally developed to provide insight to nuclear design and operation, often using conservative models to ensure reactor safety. More recently, regulators have started to accept best estimate plus uncertainty (BEPU) methods, where the conservative assumptions are replaced by uncertainty quantification and the results are bounded by a 95% uncertainty interval. Analyzing a legacy code in this way can require significant work because it is not consistent with the intended use of the code. This work demonstrates some of these issues using the interfacial mass transfer model in CTF as a specific example.

This document will first provide a brief introduction to uncertainty analysis methods. Next, the interfacial mass transfer model used by CTF is discussed. Sensitivity studies will be used to demonstrate that black box uncertainty methodologies are very limiting. Finally, an alternative approach for uncertainty analysis will be demonstrated for a single correlation in the interfacial mass transfer model.

2. Uncertainty Analysis. Uncertainty in a computational tool originates from a variety of sources, which can be grouped into four general categories: code bugs, numerical uncertainty, model form uncertainty, and parameter/correlation uncertainty.

The uncertainty from code bugs is minimized using thorough regression testing, where test problems are each designed to confirm that a very small section of code is working

*Pennsylvania State University Department of Mechanical and Nuclear Engineering, nwp110@psu.edu
†Sandia National Laboratories, vamouss@sandia.gov

correctly. To assess numerical error, code solutions are first compared to known solutions to ensure that the numerical method is correctly implemented. Then the error introduced by the mesh and time step can be quantified. If the bug or numerical uncertainty is dominant, the study of other uncertainty contributions can be meaningless. Large uncertainties can also be introduced when the fundamental assumptions of a model and a simulation are not consistent. These uncertainties are estimated by comparing code results to experimental validation tests.

Once the code has a suite of regression, verification and validation tests, the final source of uncertainty can be addressed. Parameter uncertainty deals with the error introduced to the code by using a specific correlation. This includes experimental error, any incorrect or generalized scaling arguments, and also assumptions made in the experimental setup that are inconsistent with the code. Traditionally, uncertainty quantification studies focus only on parameter uncertainty and make the assumption that the other contributions are small. This may be true in some cases, but the generalization cannot be made without first quantifying all uncertainty sources.

Previous uncertainty quantification efforts for CTF have assumed that parameter uncertainty is the only large contribution and used "black box" approaches, where the inner-workings of the code are treated as unimportant [10] [11]. One such method is the application of Wilk's formula [16], which provides the code user with a specified number of code runs to get a 95% confidence of the 95% distribution of an output parameter. This method is very enticing because it is simple, computationally inexpensive, requires no detailed knowledge of the code, and the user has freedom to set input uncertainties. This method can provide a meaningful first estimate of code uncertainty, but its use implies a number of assumptions that are not generally applicable to nuclear codes.

- The code does not crash.
- Input uncertainty distributions can be sampled.
- All input parameters are studied.
- The code has no inherent biases.

Since these assumptions are not always satisfied when using nuclear codes, the results from an uncertainty analysis using Wilk's formula are highly suspect. In addition, expert opinion is generally used to estimate parameter distributions when using Wilk's formula. This introduces a bias according to the expertise and knowledge of the selected expert. Expert opinion is generally provided as a conservatively large estimate and nonphysical combinations of parameter space can be the result. When one considers these limitations, it becomes apparent that a less simplified model of code uncertainty should be adopted.

A variety of methods have been designed to perform holistic analyses, all of which require an intimate knowledge of the code and take much more effort than black box methods. For example, the Predictive Capability Maturity Model (PCMM) [9] was recently introduced as a guide to improve modeling and simulation tools with a focus on engineering development. It addresses all forms of uncertainty discussed in this section in a thorough, step-by-step process. This work doesn't specifically follow the PCMM, but it demonstrates some of the steps and ideals.

3. CTF Interfacial Mass Transfer Closure Model. Legacy codes often have models and simplifications that are unexplained in existing documentation, and there is extreme value in reverse engineering this information. As an example, the mass transfer model in CTF will be discussed briefly; more detail will be found in the CTF Theory Manual [14]. To adequately quantify the uncertainty from a model, it is first necessary to understand it.

The mass transfer has four possible contributions: subcooled liquid (scl), subcooled vapor (scv), superheated liquid (shl), and superheated vapor (shv). The net mass transfer

is the sum of two of the contributions since each phase can either be subcooled or superheated at a given time. Each contribution is calculated by dividing the total energy transferred via phase change by the latent heat.

$$\Gamma_{net} = H_{shl}\frac{h_f - h_f^s}{C_{p,l}h_{fg}} + H_{shv}\frac{h_g - h_g^s}{C_{p,v}h_{fg}} - H_{scl}\frac{h_f^s - h_f}{C_{p,l}h_{fg}} - H_{scv}\frac{h_g^s - h_g}{C_{p,v}h_{fg}} \tag{3.1}$$

The net mass transfer is Γ_{net}, H is the heat transfer coefficient for each phase, h is enthalpy, and C_p is specific heat at constant pressure. The subscripts f and g represent the liquid and vapor phases, respectively, and the superscript s indicates an enthalpy evaluated at saturation. This model is demonstrated in Figure 3.1 for a single bubble. Superheated contributions, which cause evaporation, are shown in red and subcooled effects, causing condensation, are blue.

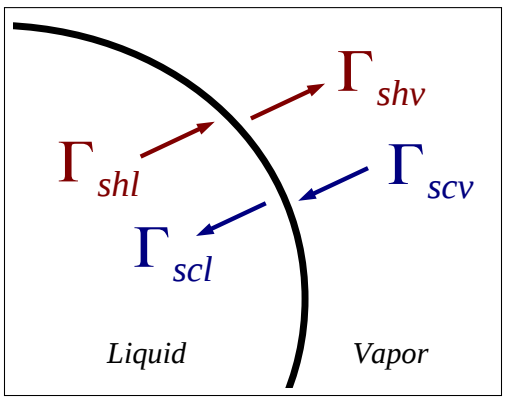


Fig. 3.1: Mass transfer model in CTF

The fluid properties are evaluated according to the state equation and the heat transfer coefficients are calculated from a variety of correlations. The correlations are averaged in state space according to a flow regime map, which is divided into four normal regimes and a single hot wall regime. When the surface temperature of a rod exceeds the wetting temperature, the hot wall regime is used, which is a combination of the hot wall droplet and hot wall film correlations. Each normal regime is some combination of three different correlations: small bubble, large bubble, and film+droplet. The normal regime map is shown in Figure 3.2.

The correlations are from various sources and a significant amount of effort was expended to find their origins. A large number are based on experimental data, such as the Lee and Ryley [8] or the Hughmark [7] correlations. Some are based on analytical solutions, such as the Boussinesq [1] correlation, which is derived for a single particle in potential flow. Still others are based purely on engineering judgment. As an example, the superheated liquid large bubble heat transfer coefficient is equal to $278.0 Btu/Ft^2hr^\circ F$, which is based on the assumption that the superheated liquid, as an unstable phase, will quickly move towards saturation. How a correlation is treated during an uncertainty analysis will depend on its origin.

When a code lacks sufficient documentation, the origin of various models and correlations can be lost. This is especially important when attempting to perform an uncertainty

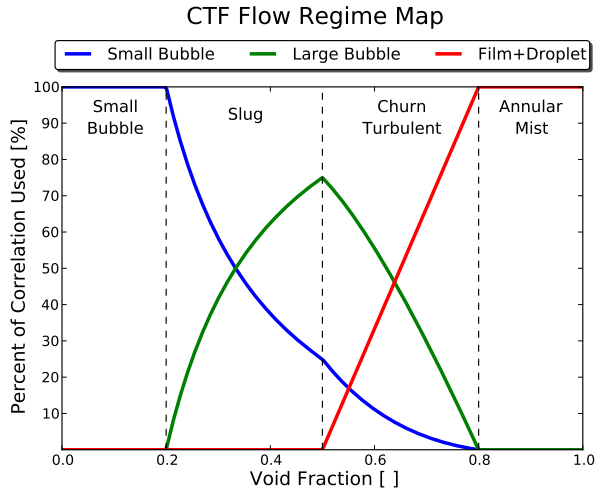


Fig. 3.2: Forced Convection Flow Regime Map

analysis because a detailed knowledge of the code is essential to the process. A few examples that demonstrate these issues are shown in Equation 3.2.

$$1,000,000 \frac{Btu}{ft^2 hr^\circ F} \approx 278.0 \frac{Btu}{ft^2 s^\circ F} \tag{3.2a}$$

$$\frac{2}{\sqrt{\pi}} \approx 1.128 \tag{3.2b}$$

$$0.738 \approx 0.74 \tag{3.2c}$$

The first example, a heat transfer coefficient of 278.0, is from the conversion of units from hr^{-1} to s^{-1}. An arbitrarily large value was selected based on engineering judgment, but the logic behind this selection was lost. The second example seems to have no physical significance in its numerical form, but it originates from an analytical derivation. The final example is simply the round-off of a value from a correlation, which is very common in CTF. All of this detailed information was lost in the pre-existing documentation and these examples clearly demonstrate the importance of retaining this information.

Another aspect of creating documentation is that it helps to locate code bugs. For example, one of the correlations is calculated based on the Jakob number, which is the ratio of sensible and latent energy absorbed during phase change. When comparing the theory manual to the source code, it became apparent that the Jakob number was calculated incorrectly. Bugs like this become obvious when the theory manual and source code are inconsistent.

CTF uses steady state, fully developed correlations to approximate transient behavior. This eliminates the length and time scales associated with phenomena, which causes discontinuities throughout the code. Smooth solutions are required to apply the numerical method, so artificial smoothing must be introduced. This includes the state space smoothing of the flow regime map, as well as spatial and temporal smoothing.

Before the heat transfer coefficients are calculated, all inputs to the model are smoothed between two adjacent axial cells. After the calculation of the heat transfer coefficients, they

are averaged again to values at the original mesh. This effectively means that the coefficient in each cell is informed by the conditions in its own cell, as well as the two surrounding cells. This ensures that mass transfer gradients between adjacent control volumes are minimized.

$$x_J = 0.5x_j + 0.5x_{j+1} \tag{3.3a}$$

$$H_j = 0.5H_J + 0.5H_{J-1} \tag{3.3b}$$

Where x is some arbitrary scalar mesh quantity, j indicates the original scalar mesh, and J indicates location on the intermediate momentum mesh. After the heat transfer coefficients are calculated, they are smoothed over time using exponential under relaxation.

$$H_k = (H_k^n)^\epsilon \cdot (H_k^{n+1})^{1-\epsilon} \tag{3.4}$$

The superscript n indicates that the coefficient is calculated at the previous time step and $n+1$ is at the current time step. The subscript k indicates the phase. The under relaxation parameter, ϵ, is 0.9 in CTF. The heat transfer coefficients are then limited to avoid any nonphysically large values. After this, the subcooled or superheated contribution for each phase is turned off.

The use of these operations is necessary for code robustness, but they can introduce uncertainties that are difficult to quantify. For example, the averaging scheme is not weighted by cell volume, which will introduce significant uncertainty when two adjacent cells have different sizes. The order of these operations can also play a significant role in code stability and uncertainty. For example, the switch from subcooled to superheated correlations occurs after the under relaxation. When the liquid is close to saturation and switches between the two contributions, the time smoothing on the heat transfer coefficient used in the calculation is effectively eliminated. A second example is that the clipping of the heat transfer coefficients to avoid nonphysically large values can completely eliminate the use of some correlations.

4. Sensitivity Studies. Once the details of the mass transfer model are fully understood, it is necessary to perform a thorough sensitivity study to determine which parameters have the most impact on a simulation. To achieve this goal, a test problem was designed that transitions through all flow regimes and uses all interfacial heat transfer correlations. The test problem models a single subchannel from a pressurized water reactor. The transient begins at nominal power and the linear heat rate is increased to 600% over 100 seconds. At the end of the transient, the exit quality is nearly unity and the transient is sufficiently long that it approximates a series of steady state calculations.

Multipliers were implemented on all interfacial heat transfer correlations in the code, as well as many of the numerical parameters discussed in the previous section. Each of the multipliers is used to independently vary the parameter or correlation from 95% to 105%. The simulation is run with various multiplier values and the change in the solution is assessed quantitatively using a nondimensional L^2, or Euclidean, norm.

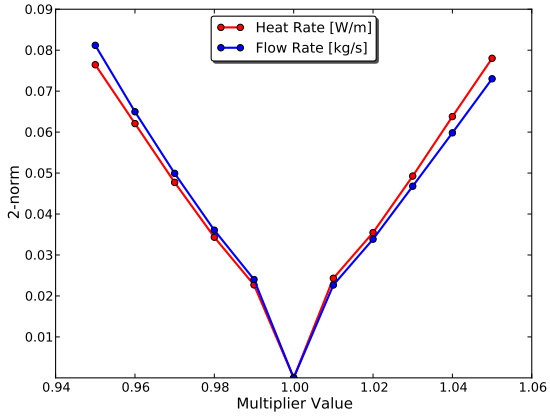
$$\|x\|_2 = \frac{\sqrt{\sum_i (x_{n,i} - x_i)^2}}{\sqrt{\sum_i x_{n,i}^2}} \tag{4.1}$$

Where x_i is the exit quality at time step i with the multiplier and $x_{n,i}$ is the nominal value at that time step. This essentially measures the "distance" from the nominal run

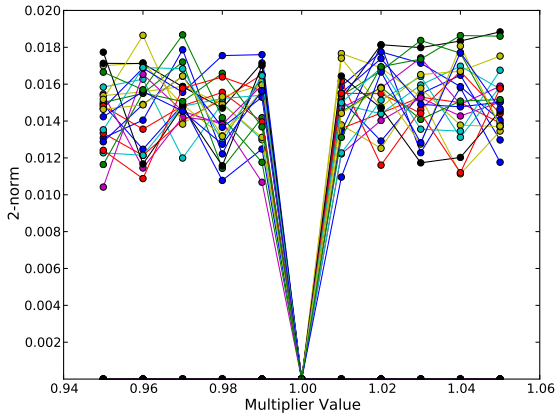
caused by a perturbation in the multiplier. This distance can be plotted with respect to the multiplier value. The expected behavior is shown in Figure 4.1(a), where the distance from nominal increases as the multiplier gets further from unity. This is described by a characteristic V-shape, which is shown for two of the inputs with the largest impact on the exit quality: the mass flow rate and the heat rate.

When the norms are plotted as a function of the correlation multiplier, the distance from nominal has no relationship to the initial perturbation, which does not follow the expected behavior. This is because the "signal" that is being measured, correlation sensitivity, is much smaller than the "noise", this is shown in Figure 4.1(b). Note that the simulation is identical to nominal when the multiplier is equal to one. The noise can come from numerical error and the discontinuities caused in the flow regimes, but in this case, it can easily be eliminated by lowering the time step by a factor of 1000. The results are shown in Figure 4.1(c) for the subcooled small bubble correlations. The sensitivity of the solution to the correlations is about 0.003, while the numerical noise is about five times larger. This process is important because it ensures that the solution is behaving as expected and that the desired sensitivities are measured.

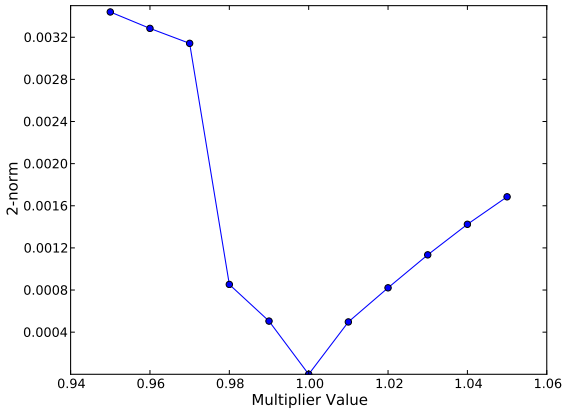
The maximum distance from nominal for each multiplier could be used to give a quantitative measure of the parameter's overall influence on the simulation. This information is important because the parameters with the highest sensitivities are likely to impart the most uncertainty. The simulations with the small time step are necessary to reveal true sensitivity information from the selected test problem. Figure 4.1 demonstrates that the numerical uncertainty is about five times larger than the parameter uncertainty of the bulk mass transfer model, so future work must focus on reducing the numerical error before further investigating the mass transfer uncertainty. Because the parameter uncertainty is hidden by the numerical uncertainty for the mass transfer model, assessment of the parameter sensitivity would be a purely academic exercise and is not performed here.



(a) Parameter uncertainties dominant, Δt=2E-2



(b) Numerical uncertainties dominant, Δt=2E-2



(c) True sensitivity with small time step, Δt=2E-6

Fig. 4.1: Sensitivity Study

5. Bayesian Calibration of the Lee and Ryley correlation. This section will demonstrate the minute detail necessary to perform a holistic uncertainty analysis. Each part of every model must be explored in depth and the uncertainty quantified. The Lee and Ryley correlation [8] will be used as an example. This correlation is used in CTF for the vapor heat transfer coefficient for large bubbles and droplets, as well as for the superheated small bubble correlation. It is one in a family of correlations that take the same form.

$$Nu = 2.0 + ARe^{1/2}Pr^{1/3} \tag{5.1}$$

The Nusselt number is defined as $Nu = hD/k$, Reynolds number as $Re = \rho v D/\mu$ and the Prandtl number as $Pr = C_p\mu/k$. The velocity of the continuous phase is v, h is the heat transfer coefficient, D is the particle diameter, the thermal conductivity and specific heat of the continuous phase are k and C_p, respectively. The coefficient A is determined experimentally and the other coefficients do not change. This form first appeared in 1938 with a coefficient of 0.55, where it was derived for mass transfer from droplets to air [3]. The second proposed form used a coefficient of 0.60 [12], and the form that is examined here uses a coefficient of 0.738 and is determined from experiments of water droplets in steam.

To calibrate this correlation, a total of 1233 data points were gathered from 24 different sources for heat and mass transfer to single spheres [13] [6]. The data is grouped into four categories according to the dispersed and continuous phases. The dispersed phase is either solid or liquid and the continuous phase is either liquid or air. The dependence of Nu on Re, Pr and $Re^{1/2}Pr^{1/3}$ across this data set is shown in Figure 5.1. Note that the Lee and Ryley correlation is shown as a dashed line in Figure 5.1(c).

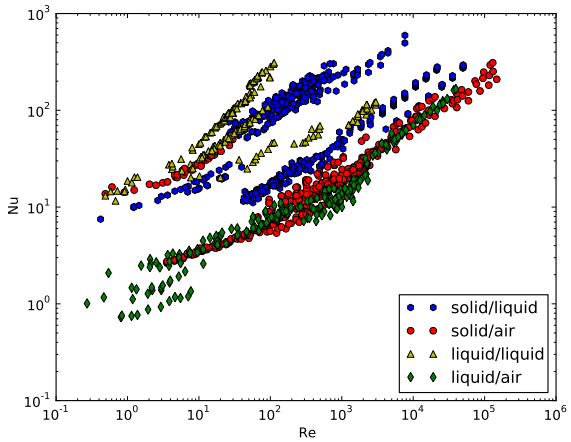
The large majority of the data is for heat transfer from solid spheres to air at atmospheric conditions. This introduces a model form uncertainty because the model is applied to water in steam or steam in water. This is not ideal, but experiments with superheated steam present significant difficulties. Instead, scaling arguments are made to put the correlation in nondimensional space and then it is applied to reactor conditions. This process assumes that the correct functional form is captured in Equation 5.1.

Outliers are observed at low $Re^{1/2}Pr^{1/3}$ in Figure 5.1(c). These are from the experiments of Yuen and Chen, which had significant temperature differences between the continuous and dispersed phases [17]. They proposed a factor to account for the effect of superheat. Similarly, the highest values of the liquid/liquid data are under-predicted by the Lee and Ryley correlation. This is because the experiments of Griffith have significant differences in the viscosity of the continuous and dispersed phases [5]. In addition, natural convection can become important at low velocities, which can be accounted for by the inclusion of the Grashof number, Gr. This suggests that the functional form of the correlation should have more than two dimensions.

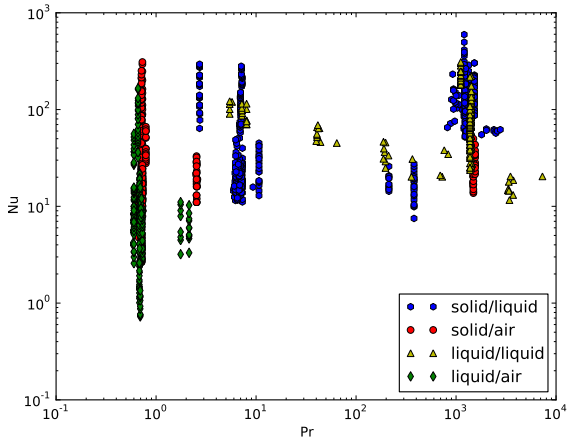
$$Nu = f(Re, Pr, \frac{\Delta h}{h_{fg}}, \frac{\mu_c}{\mu_d}, Gr) \tag{5.2}$$

The viscosity is μ and the subscripts c and d represent the continuous and dispersed phases, respectively. For this work, it is assumed that the functional form given in Equation 5.1 is correct and that inclusion of data sets with the higher dimensional dependence will account for this assumption in the final uncertainty results. Therefore, the form of the correlation that will be calibrated has four coefficients: θ_1, θ_2, θ_3 and θ_4.

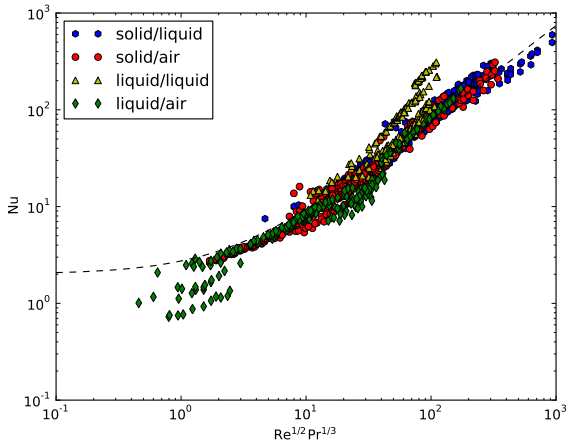
$$Nu = \theta_1 + \theta_2 Re^{\theta_3} Pr^{\theta_4} \tag{5.3}$$



(a) Dependence on Reynolds Number



(b) Dependence on Prandtl Number



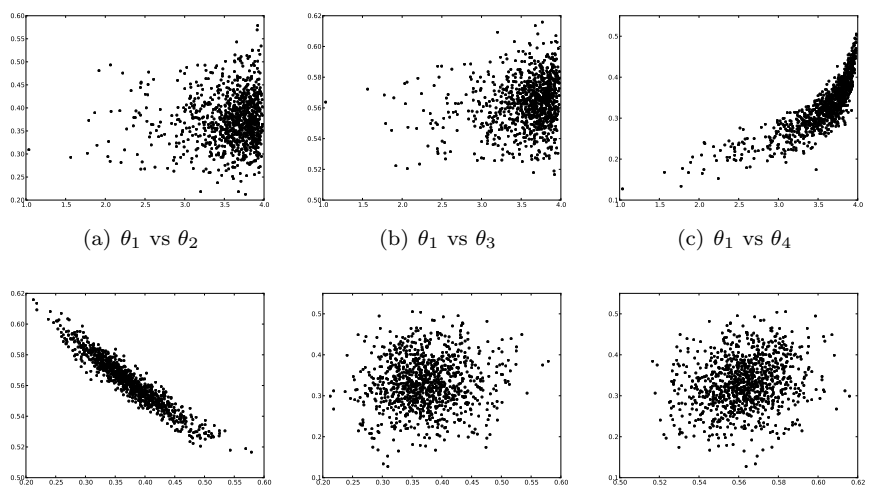
(c) Plot of the raw data against Lee and Ryley correlation

Fig. 5.1: Raw data used in Bayesian calibration

An additional consideration is that there are a number of different definitions of the nondimensional numbers, all of which are equally valid. For example, some experimentalists evaluate properties at the surface of the particle, while others use the free stream numbers. This adds ambiguity to the analysis, which will be neglected here.

These coefficients are calibrated to experimental data using Bayesian methods. This allows for the explicit treatment of the experimental uncertainty, which is used to obtain information about the marginal and joint distributions of the four coefficients. This method provides extremely detailed uncertainty information, with realistic values for the coefficients and reduced uncertainty when compared to expert opinion. It also easily allows for additional experimental data points to be included and the results re-calibrated. For brevity, the details of Bayesian methods are not discussed here; refer to [4] for more information.

Each of the four sets of data is calibrated separately. A joint distribution between the four coefficients is created from the experimental data and then sampled 1000 times. The sampled data is used to gather information about the coefficient distributions. The sampled points for the solid/air data are shown in Figure 5.2, where the joint distribution has been represented as six bivariate plots. Some of the coefficients are strongly correlated, for example, there is a strong correlation between θ_1 and θ_4, as well as between θ_2 and θ_3. Most black box methods assume that input uncertainties are independent, so this is one of the many advantages of using Bayesian calibration.



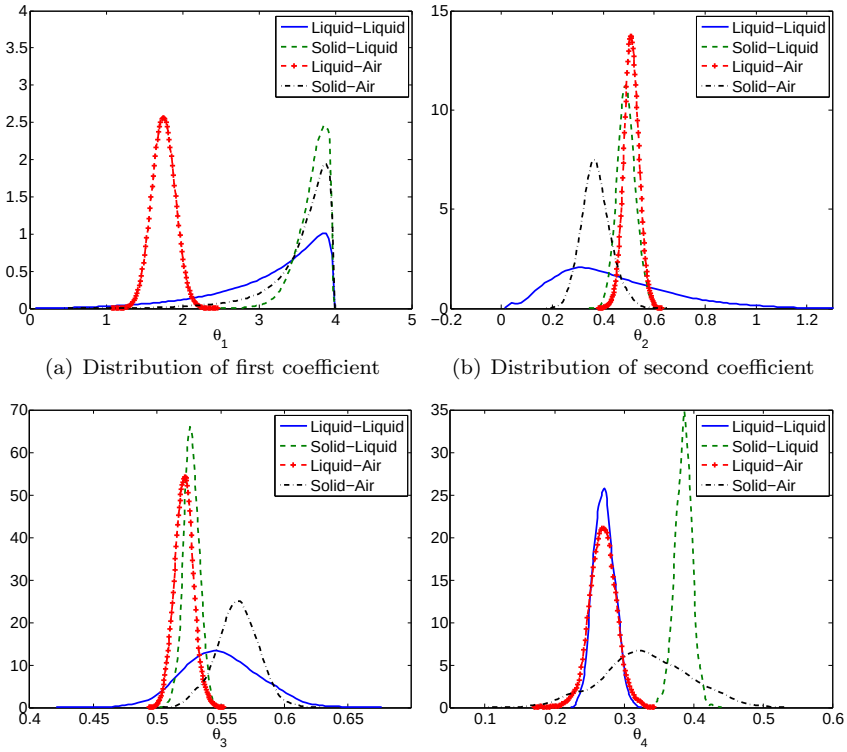
(a) θ_1 vs θ_2 (b) θ_1 vs θ_3 (c) θ_1 vs θ_4

(d) θ_2 vs θ_3 (e) θ_2 vs θ_4 (f) θ_3 vs θ_4

Fig. 5.2: Bivariate projections of data sampled from solid/air joint distribution

Using the 1000 data points, marginal distributions can be constructed. These distributions will lose information about the correlation between the coefficients and instead give information about the uncertainty of each coefficient independent of the others. This is useful because it makes comparison to traditional distribution concepts, such as mean and standard deviation, much easier. These results are shown in Figure 5.3 for each of the four material combinations. The results for the four coefficients are very different depending on the experimental data used to calibrate the correlation.

This information can be summarized using a simple mean and standard deviation.



(a) Distribution of first coefficient (b) Distribution of second coefficient

(c) Distribution of third coefficient (d) Distribution of fourth coefficient

Fig. 5.3: Marginal distributions from Bayesian calibration

$$\mu = \frac{1}{N}\sum_{i=1}^{N} x_i \tag{5.4}$$

$$\sigma^2 = \frac{1}{N-1}\sum_{i=1}^{N} (x_i - \mu)^2 \tag{5.5}$$

The results for all material combinations are shown in Table 5.1. These results show large differences in the distribution depending on which experimental data is selected.

The only coefficient that is within one standard deviation of the value used by Lee and Ryley is the Prandtl exponent for the solid/air experiments. The only results within two standard deviations is in the liquid/liquid coefficients, which have large uncertainties and are therefore questionable. All other results vary from the expected values by up to nine standard deviations. Additionally, θ_1, which has been given both theoretical [1] and experimental [2] backing, varies significantly from the value used in the correlation.

All of these issues raise important questions about the applicability of the assumptions used to make the correlation and also the applicability of the correlation within CTF. Many of the issues discussed earlier may have an impact on these results. The functional form of the correlation does not include superheat, viscosity, or natural circulation, and there may

Dispersed/ continuous	θ_1 *(2.0)*		θ_2 *(0.738)*		θ_3 *(1/2)*		θ_4 *(1/3)*	
	μ	σ	μ	σ	μ	σ	μ	σ
Liquid/air	1.752	0.097	0.511	0.028	0.521	0.007	0.270	0.019
Liquid/Liquid	3.244	0.577	0.428	0.207	0.550	0.029	0.271	0.015
Solid/air	3.565	0.390	0.373	0.055	0.563	0.016	0.331	0.063
Solid/Liquid	3.706	0.183	0.491	0.034	0.527	0.006	0.386	0.012

Table 5.1: Mean value and standard deviation for all four coefficients for each material combination

be additional dimensions. The experimentalists used different forms of the nondimensional numbers, which introduces a large amount of uncertainty. Finally, the fundamental assumptions between different experiments may be inconsistent. More work would be necessary to draw conclusions about the uncertainty of the Lee and Ryley correlation, but this work provides the first step.

6. Conclusion. This work has demonstrated some of the difficulties presented when performing VVUQ on legacy codes. Many older codes have limited documentation, which is the first difficulty in this process. Additionally, it can be difficult to measure uncertainties and sensitivities when they are hidden by noise, which originates from other sources of uncertainty. In CTF, the noise can be eliminated by running the problem at extremely small time steps, but this will not be a solution for all codes. Once the sensitivities are measured correctly, the correlations and numerical parameters can be ranked according to their importance to the problem.

In the CTF mass transfer model, it is shown that uncertainties may originate from numerical smoothing parameters, which are necessary because the code uses steady state fully developed correlations to approximate transient behavior. If these numerical parameters contribute significantly to the code uncertainty, black box uncertainty analysis methods, which ignore these numerical parameters, are extremely limiting. Nonetheless, it is also clear why these uncertainties are so often neglected.

It is difficult to determine the distribution of numerical smoothing parameters, values based on engineering judgment, or values from analytical derivations. For example, it is difficult to create a justifiable distribution for the under relaxation parameter (ϵ in Equation 3.4) because numerical issues will be caused if it is too small. On the other hand, the distribution cannot be based on code robustness because that would not be a true measure of uncertainty. Another example is the large values selected for the heat transfer coefficients that quickly force the unstable phases towards saturation. The meaning of "large" is very subjective and it is difficult to specify its distribution. Methods for defining these distributions will be necessary to adequately quantify the uncertainty of the rest of the mass transfer model.

The uncertainty of correlations that are based on experimental data can be thoroughly investigated because it is clear how the experimental data can inform the distribution of the correlation coefficients. To demonstrate this process, Bayesian calibration was performed on the Lee and Ryley correlation. This process allows experimental uncertainty from multiple sources to directly inform the correlation uncertainty. These uncertainties can then be used as justifiable distributions during code simulations. This process is extremely robust and flexible.

This work could be completed by first quantitatively ranking the correlations and nu-

merical parameters in the mass transfer model. This information can then be used to focus uncertainty quantification efforts on the most influential parameters. The correlations can be analyzed using Bayesian calibration, like with the Lee and Ryley correlation in this work. Methods for assigning input uncertainty distributions to smoothing parameters, analytical derivations, and values based on engineering judgment must also be developed to complete this work.

Acknowledgments. The author would like to thank Robert Salko (ORNL) and Maria Avramova (PSU) for their extensive knowledge of CTF and their availability to discuss issues relevant to this paper. Additional thanks to Russell Hooper (SNL), who provided a very helpful review, and to Brian Williams (LANL), Ralph Smith (NCSU), and Allison Lewis (NCSU) for their assistance with the Bayesian calibration process. This research was supported by the Consortium for Advanced Simulation of Light Water Reactors (http://www.casl.gov), an Energy Innovation Hub (http://www.energy.gov/hubs) for Modeling and Simulation of Nuclear Reactors under U.S. Department of Energy Contract Number DE-AC05-00OR22725.

REFERENCES

[1] M. J. Boussinesq, *Calcul du pouvoir refroidissant des courants fluids*, Journal de Mathématiques Pures et Appliquées, 1 (1905), p. 310.

[2] S. K. Friedlander, *Mass and heat transfer to single spheres and cylinders at low reynolds numbers*, AIChE Journal, 3 (1957), p. 43.

[3] N. Frössling, *Uber die verdunstung fallender tropfen (The evaporation of falling drops)*, Gerlands Beitrage zur Geophysik, 52 (1938), pp. 170–216.

[4] A. Gelman, J. B. Carlin, H. S. Stern, and D. B. Rubin, *Bayesian Data Analysis*, Chapman & Hall/CTC, 1997.

[5] R. M. Griffith, *Mass transfer from drops and bubbles*, Chemical Engineering Science, 12 (1960), pp. 198–213.

[6] G. A. Hughmark, *Mass and heat transfer from rigid spheres*, AIChE Journal, 13 (1967), pp. 1219–1221.

[7] M. Ishii and M. A. Grolmes, *Inception criteria for droplet entrainment in two-phase concurrent film flow*, AIChE Journal, 21 (1975), pp. 308–318.

[8] K. Lee and D. J. Ryley, *The evaporation of water droplets in superheated steam*, Journal of Heat Transfer, 90 (1968), pp. 445–451.

[9] W. L. Oberkampf, M. Pilch, and T. G. Trucano, *Predictive capability maturity model for computational modeling and simulation*, Tech. Rep. SAND2007-5948, Sandia National Laboratories, October 2007.

[10] Y. Perin et al., *Uncertainty analysis of CTF prediction of moderator and fuel parameters for the OECD LWR UAM benchmark using exercise II-3*, NURETH-15, (2015), pp. 12–15.

[11] N. Porter, M. Avramova, and K. Ivanov, *Uncertainty and sensitivity analysis of COBRA-TF for the OECD LWR UAM benchmark using Dakota*, NURETH-16, (2015). to appear.

[12] W. E. Ranz and W. R. Marshall, *Evaporation from drops*, Chemical Engineering Progress, 48 (1952), pp. 141–146, 173–180.

[13] P. N. Rowe, K. T. Claxton, and J. B. Lewis, *Heat and mass transfer from a single sphere in an extensive flowing fluid*, Transactions of the Institution of Chemical Engineers, 43 (1965), pp. T14–T31.

[14] R. K. Salko and M. N. Avramova, *CTF Theory Manual*, Penn State, May 2015. to be editted.

[15] M. J. Thurgood et al., *COBRA/TRAC - A thermal-hydraulics code for transient analysis of nuclear reactor vessels and primary coolant systems*, Tech. Rep. NUREG/CR-3046, PNL-4385, US Nuclear Regulatory Commission, 1983.

[16] S. S. Wilks, *Determination of sample sizes for setting tolerance limits*, The Annals of Mathematical Statistics, 12 (1941), pp. 91–96.

[17] M. C. Yuen and L. W. Chen, *Heat-transfer measurements of evaporating liquid droplets*, International Journal of Heat and Mass Transfer, 21 (1978), pp. 537–542.

SPPARKS SOFTWARE UPDATES

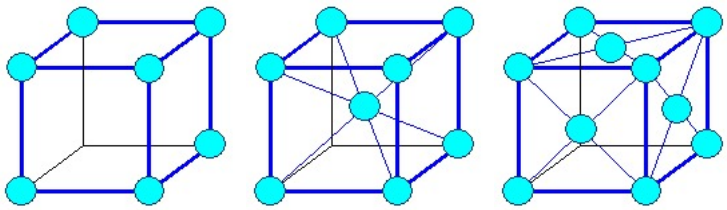
JUSTIN M. ROBERTS*, JOHN A. MITCHELL†, AIDAN P. THOMPSON‡, AND VEENA TIKARE§

Abstract. SPPARKS, an acronym for Stochastic Parallel Particle Kinetic Simulator, is a kinetic Monte Carlo algorithm to simulate events such as grain growth and diffusion. New applications for SPPARKS have been developed but not fully tested or documented. Our work was to understand how SPPARKS operates in order to finish testing and documenting new applications of SPPARKS. This report outlines what was accomplished during the summer of 2015.

1. Introduction. After we learned how to compile and run SPPARKS, we were able to start testing and writing documentation for new SPPARKS applications. This required updating the new applications to the latest version of SPPARKS. During the summer of 2015, we were able to completely finish the Potts gradient application and made significant progress on the Curvature diagnostic.

2. Understanding SPPARKS terminology. SPPARKS uses its own software specific terminology to refer to particular data structures and operations. We will explain some of the terminology here.

SPPARKS sets up sites to run simulations. A site is a point in the lattice at which an event can occur. The events performed at a given site are unique to the application driving SPPARKS. For example, below we make mention of the Potts gradient application. Events on sites in a Potts gradient application refer to the site flipping its spin, or in simple terms switching to a different grain. Sites in a simulation are situated in a specified pattern or lattice. A few examples of lattice patterns are square, simple cubic, face-centered cubic (fcc), body-centered cubic (bcc), and diamond.



(a) Simple Cubic Lattice (b) BCC Lattice (c) FCC Lattice

Fig. 2.1: Example Lattice Orientations[1]

3. Understanding SPPARKS. SPPARKS is very complex code that can be run in parallel or serial. It is controlled using an input file which has various different commands provided by the user[3]. A few examples of SPPARKS commands are *sweep*, *lattice*, *solve_style*, *app_style*, *run*, and *diag_style*. These commands specify which SPPARKS application to use, which diagnostic tools to use, and how it is solved. The input file is read with the parse function called from the *file()* function within *input.cpp*. After it is parsed, the

*Dept. of Mechanical Engineering, Brigham Young University, justinrobertsdw@gmail.com
†Sandia National Laboratories, jamitch@sandia.gov
‡Sandia National Laboratories, athomps@sandia.gov
§Sandia National Laboratories, vtikare@sandia.gov

command is executed from a list of options within the *execute_command()* function within *input.cpp*. The *iterate()* function controls which solver to use. The simulation can be solved by using either KMC (Kinetic Monte Carlo) or rKMC (rejection Kinetic Monte Carlo). The *iterate()* function is found from within *app_lattice.cpp* and is called from *run()* found in *app.cpp*. The *run* command specifies how many time steps the program will execute.

SPPARKS generates information from a simulation in various forms. Information can come from the log file, dump images, dump files, and diagnostics. Diagnostics can be chosen using the *diag_style* command. We are currently working on a new *diag_style* command called *curvature*.

4. Curvature. The Curvature diagnostic was originally developed by Veena Tikare (vtikare@sandia.gov). The diagnostic computes curvatures for each grain and then writes this information to disk for further analysis. It also has the option of writing the number of grains to the log file at a given time step defined by the *stats* command. The log file option was not originally part of the code but was added by using the functions *stats_header()* and *stats()*. These functions are called from the functions *stats()* and *stats_header()* from *output.cpp* in the SPPARKS source code. An example of how a log file is output to the screen during run time is given below. Notice that Ngrains is an additional column provided by the *curvature* diagnostic Table 4.1.

Curvature M_v is defined as

$$M_v = \sum_{edges} \frac{1}{2}\beta L,$$

where L is the length of each edge, and β is the angle formed by two faces on an edge. After implementing the function into SPPARKS it becomes:

$$M_v = \frac{1}{2}\frac{\pi}{2}(E_o - E_i)L,$$

where E_o is the number of edges on a face which form an angle of $\pi/2$. E_i is the number of edges on a face which form an angle of $-\pi/2$. Grain edges that share three different grains and grain corners are not counted as part of the face curvature.

The algorithm is still being tested but is very close to being added to the SPPARKS repository and included as a diagnostic option of SPPARKS.

Time	Naccept	Nreject	Nsweeps	CPU	Energy	Ngrains
0	0	0	0	0	406250	15625
2.5	93493	922132	65	0.518	127208	108
5.03846	109515	1937360	131	1.01	101282	51
7.53846	121225	2941275	196	1.49	89664	35
10.0385	132541	3945584	261	1.98	77620	24
12.5385	141994	4951756	326	2.45	69984	18
15.0385	151674	5957701	391	2.93	60976	15
17.5	162959	6946416	455	3.4	43178	9
20	171903	7953097	520	3.86	17004	2
22.5	172496	8968129	585	4.31	16600	2
25	172972	9983278	650	4.76	16600	2

Table 4.1: Command line output with Ngrains

5. Potts Gradient. Another application of SPPARKS we worked on and completed was an application called Potts gradient. The Potts model is typically used to simulate grain growth in metals. Grain growth occurs when a metal is held at a high temperature; at high temperature the metal undergoes recovery, recrystallization, and nucleation.

The Potts gradient application adds temperature gradients to the Potts model. Every site in the model is assigned a temperature given by a linear function. The linear function is uniquely defined by the value T_0, at the center of the lattice, and gradients in the x,y, and z directions.

The equation

$$M_0 e^{\frac{-Q}{KT}}$$

defines the mobility at each site where M_0 is the mobility constant, K is Boltzmann's constant ($8.6171 \times 10^{-5} \frac{eV}{K}$), T is the temperature of the site, and Q is the activation energy. The grain boundary mobility affects the probability that a particular grain will grow. Higher mobility means higher probability of grain growth. The algorithm was originally intended for use with temperature gradients, but we later added mobility gradients as another option. When mobility gradients are used, each site is assigned a mobility. The mobility is initialized at the center of the lattice and then is assigned to each site depending on the site's position in the lattice. The mobility gradient is also defined as a linear function, analogous to the described above for temperature.

The code was extensively tested and released with the SPPARKS source code. Testing required image generation. A program was used to translate SPPARKS dump files to *paraview* image files. Paraview enabled us to view grain growth animations. With this tool, we were able to uncover a critical bug in the code that was allowing negative temperatures and mobilities to be assigned to certain sites. The user is expected to provide reasonable gradients that don't cause negative temperatures or mobilities. If the gradients cause negative values, the program terminates with an error statement.

Figure 5.1 demonstrates the mobility gradient option of the Potts gradient model. As shown, grains on the left are much larger than the grains on the right. This is because mobility on the left is significantly larger than mobility on the right.

To generate Figure 5.1, an initial mobility of 0.5 was chosen at the center with lattice dimensions 400 X 100 X 100 giving a total of 4,000,000 sites oriented on a simple cubic lattice. A mobility gradient of .0025 was defined in the X direction which gives a mobility

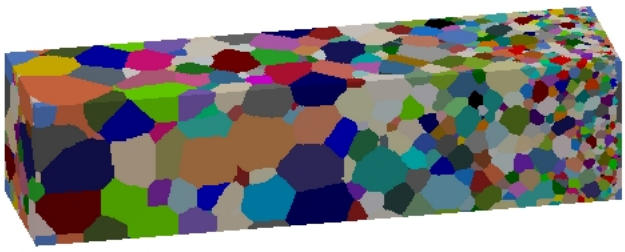


Fig. 5.1: Potts model with mobility gradients applied

of 1.0 at x-max and 0 at x = 0. This example illustrates the affect of mobility gradient on grain growth; a temperature gradient has a similar effect[2]. These parameters display the major changes that grain growth undergoes when a mobility gradient is applied. We also generated videos of the simulation of Figure 5.1 which are posted under the pictures and movies section on the SPPARKS web site[3].

6. Understanding documentation. All of the HTML code on the SPPARKS web site was generated from a text to HTML converter. The converter uses markup commands to convert .txt files into HTML code that is read by a browser. The markup commands include some but not all of the options of HTML. It includes elements such as input, br, p, hr, a, ul, li, and pre. We learned how to write markup files and convert them into HTML for the SPPARKS website. We wrote documentation for the new Potts gradient application mentioned above which required updates to existing pages and a whole new doc page. The formating style needed to be consistent with other doc pages and needed careful consideration.

7. Conclusions. We were able to implement new applications into the existing SPPARKS code along with documentation provided on the SPPARKS website[3]. This required us to learn how SPPARKS operates and how to write markup code. It also required a sound understanding of material science, and computer science. More applications are in process and will be added to the SPPARKS source code and doc pages in the near future.

REFERENCES

[1] H. Foll, *Lattice and crystal*. http://www.tf.uni-kiel.de/matwis/amat/def_en/kap_1/basics/b1_3_1.html.

[2] J. A. Mitchell and V. Tikare, *Numerical simulation of ni grain growth in a thermal gradient*, SIAM Conference on Computational Science and Engineering, Salt Lake City, Utah, USA, 2015, Sandia Technical Report: SAND2015-1665c.

[3] S. Plimpton, C. Battaile, M. Chandross, L. Holm, A. Thompson, V. Tikare, G. Wagner, E. Webb, X. Zhou, C. G. Cardona, and A. Slepoy, *Spparks kinetic monte carlo simulator*. http://spparks.sandia.gov/index.html.

[4] ———, *Crossing the Mesoscale No-Man's Land via Parallel Kinetic Monte Carlo*, Sandia report SAND2009-6226, October 2009.

GRAPH REPRESENTATION FOR NEURAL NETWORKS

FELIX WANG* AND FRED ROTHGANGER†

Abstract. How we formulate and define a problem that is critical to our ability to manipulate the relevant ideas in thinking about that problem. In the domain of neural networks, in particular, spike timing based models, there is a need for a suitable representation that incorporates both state dynamics as well as structural dynamics. As a practical matter, the usability and scalability of the representation as mapped onto an implementation in software is also important. We introduce a graph-theoretic approach to representation in order to address these needs, both as a way of specifying the construction of a complex neural network and in terms of describing its time evolution during simulation.

1. General Overview. The goal of providing a graph-theoretic description of a neural network is to facilitate the efficient and straightforward exchange of network models [5, 3]. There are five major *components* that make up the graph representation language designed to describe and construct neural networks. These are, in order of dependency: *vertex*, *edge*, *graph*, *subgraph*, and *association*.

Generally, a *graph* is composed of *vertices* and *edges*, $G(V, E)$, where V is the vertex set and E is the edge set. To accommodate the construction of complex graph connectivity, we introduce the concept of *subgraphs* and their *associations*. A subgraph, as its name implies, contains a subset of the vertices and edges of a graph. If a subgraph shares the same vertex set as a graph, it's referred to as a spanning subgraph of that graph. If a subgraph shares the same edge set as a graph over a vertex set of that graph, it's referred to as an induced subgraph of that graph by that vertex set. The term *association*, or association scheme, is used to determine how the vertices of two subgraphs are connected, expanding upon the notion of edges. Whereas vertices are connected by edges, subgraphs are associated by schemes. Furthermore, associations may be either undirected or directed according to the underlying edges.

The role of a graph, then, is to define the vertex sets, and then to determine the relevant edges, subgraphs, and associations on top. Importantly, all subgraphs and associations are decomposable in terms of their vertices and edges. For graph hierarchies, or graphs of graphs, the convention we use is to treat a graph as a subgraph in a higher-order graph. The viewpoint we adopt can be thought of as vertex-centric. That is, the main actors are the vertices, with the edges coming into play when there is need for communication among vertices. That being said, there is considerable communication, and edges have a fairly substantial role in determining state dynamics as a result of information exchange.

1.1. Specification vs. Simulation. In terms of simulation, graphs specify what get simulated, being resolved down to the level of vertices and edges. In line with this, graphs (and subgraphs) don't have state dynamics associated with them. Rather, (sub)graphs provide a means to simulate structural dynamics, how the vertices and edges are organized and how they evolve throughout simulation. For (sub)graphs that may benefit from having state, such as in cases of top-down control, the procedure is to define a vertex over over the subgraph, with the connections determining the particular organization. This design principle is taken from the philosophy that *more is different*, where hierarchical levels of organization each require their own abstraction and language to reason about effectively [2]. As a result, the specification of a vertex entails an instance that is simulated by a given simulation tool, and the specification of a graph governs the existence of those instances.

*University of Illinois at Urbana-Champaign, fywang2@illinois.edu
†Sandia National Laboratories, frothga@sandia.gov

On the simulation of edges, the graph specification tends towards not creating separate instances per edge, even though edges may contain their own state dynamics (e.g. plastic synapses). Particular dynamics may require state from the vertices that are connected (e.g. voltage-dependent synapses), which poses an issue to the simulator on the locality of storage of state information. For directed edges, connecting a source to a target, it is typically the target that is more closely bound to the computation of edge state, and the composition of an edge with its target vertex would yield more efficient simulation. For undirected edges, or where information is shared equally between vertices, a shared memory model, perhaps with ghost regions across compute nodes, may be more appropriate, again incorporating edge dynamics into that of the vertex. Of course, edges may also be simulated as their own instances, more of as a fall-back mechanism.

2. Components. Although the different components types are given as *vertex*, *edge*, *graph*, *subgraph*, and *association*, further specification to handle things like I/O or expected connection behavior also exist. This additional information is provided as an *attribute* list. Other forms of metadata about a component is its *name* or identifier that may be referred to by other components (cross-file), a *description* or commentary about the particular component, and *references* of publications that a component was defined from. At the top of a component specification, various constants and equations that are used in the construction of that component but not necessarily required during simulation may be *defined*. Examples of this may include useful mathematical constants or special functions to compute spatial distribution.

2.1. Vertex. Vertices are the main instantiated object and contain information about its dynamics in the form of *state*, *parameters*, *events*, and their *equations*. The difference between *state* and *parameters* in the specification indicate whether a value should be stored (and updated) for each *instance* of a vertex *population*. Here, a population refers to a vertex set of the same vertex class (identical equations and parameters values), and is a special case of a subgraph. A population may contain several instances of a vertex, each with its own state, but a population will always contain one copy of the parameters, which are shared among all instances of that population. There may be many populations, differentiated by *name*, of vertices that are constructed *from* the same basic vertex class, each one containing a modified set of parameters (additional information in sec. 2.1.2). These parameters are evaluated once at the creation of the vertex population, while vertex states, while *initialized*, are updated according to the equations throughout the simulation.

Equations can be broken up into four basic types: *constant*, *closed-form*, *drift*, and *diffuse*. Although the terms *constant* and *closed-form* are easily understood equation types, the terms *drift* and *diffuse* may be less familiar. These terms come from the language of stochastic differential equations, where equations of state can be distinguished between a drift and diffusion process [6]. A prime example of this is the modeling of Brownian motion: a particle normally moves at a given velocity unhindered (drift), but occasionally, it will collide with another particle and its velocity changes considerably (diffuse). Whereas the drift equations are continuously integrated, the diffuse equations are evaluated as a result of events. This may be formulated as eq. 2.1.

$$dx_i = f_i(\boldsymbol{x})dt + \sum_{m=1}^{n} g_i^m(\boldsymbol{x})dN_m \tag{2.1}$$

where f gives the more common ordinary differential equation of state $\boldsymbol{x} = \{x_i | i = 1, \ldots, k\}$, and g is evaluated on $\boldsymbol{x}$ according to a counting process N for any number of events n.

The convention of order of evaluation of the different types of equations is as follows. During the initialization of a vertex instance, the constant valued parameters or state are

evaluated and remain fixed for the duration of the life of that instance in the case of state, or the life of the vertex population in the case of parameters. In the latter case, because parameters are shared among all vertices of the same group, storage in memory may be performed more efficiently. During each evaluation of a time-step or increment, the closed-form equations are evaluated at the beginning, basing their computation off of values in a previous step. Subsequently, drift equations are integrated for that time-step, which may be fixed or variable, depending on the ODE solver used. Finally, any events that were generated as a result of the integration are processed. These are treated similarly to events that were 'received' on the event queue during that time-step.

2.1.1. Event Queue. More of an implementation rather than specification detail, the event queue is simply where a vertex stores a list of events that it must evaluate at some point. These events may be generated by a process in the vertex (e.g. spiking activity), or received through an edge from another vertex. Events are indexed by both the *class* of event that occurred and a *timestamp* of when it should be evaluated. Events may not need more information than simply the class and the timestamp, such as in the case of a spike where the response dynamics can be defined at the target vertex. However, for events that require additional data to be passed between vertices, a *buffer* along with its *size* should be specified.

By convention, events are not processed until after they would have occurred. This allows for the ability to perform precisely timed events if desired by rolling back the *drift* computation to the timestamp, computing the event, and returning to the present time. Related to this, events that occur as a result of the *drift* computation (e.g. spiking reset) may occur immediately after the event conditions are met. However, it may be preferable to perform such a check after a time-step has been fully computed as to prevent excess computation, as events are sparse.

2.1.2. Inheritance and Inclusion. Following the DRY (don't repeat yourself) principle, as well as following the N2A language, the definition of vertices accommodates the ideas of inheritance from object-oriented programming [8]. Inheritance is when a vertex class acts as a specialization on top of an existing *parent* or vertex that it is based off of. All of the state, equations, etc. are *inherited* or copied from the parent, and the *child* vertex is then able to add or modify these. An especially apt use-case for this is when a parent vertex class may take on a variety of parameterizations, while maintaining the same state dynamics (dependent on these parameterizations).

Inclusion is when a vertex class *includes* or contains other vertex classes as part of its own specification. Here, the state dynamics of the included vertex is combined with that of the container vertex. An example of when this may be when an otherwise passive membrane includes a variety of ion channels. Unlike the specification of a graph (sec. 2.3), however, inclusion of vertices in a vertex does not permit connecting the included vertices through edges. Rather, the included vertices should be thought of more as modular components of that vertex. Equations in an included vertex that explicitly refer to states in a container vertex for their computation (e.g. an ion channel) should use the term *sup* (short for superset) to specify such a state.

2.1.3. Vertex Types. The three main vertex types are: *simulated*, *device*, and *file*. For all practical purposes, all models with state dynamics that exist in the graph will be of the simulated variety. These compose the various neuron classes, but may also include vertices that correspond to a particular subgraph. The standard language for simulated vertices do not work as well for describing devices or files, however. Due to their nature, these vertex types must define the *reading* and *writing* of *data* to and from something that

isn't implemented by the simulator. Moreover, the data received or to be provided may require additional processing before crossing the gap, providing something of a distinction between *data* and *state*. Devices in particular may vary greatly depending on the hardware communication protocol, making generalization difficult.

Devices correspond to external hardware (or virtual) sensors or motors that the simulation has access to and receives or transmits data, respectively. In general, *data* may be exchanged by providing a *port* to connect to by the device vertex. The expectation is that the hardware device will supply data on this port, from which the vertex in simulation is able to process before passing it to the rest of the simulation. Data may be exchanged either at a specified *rate*, in which the port is checked periodically, or provided through a *callback*, in which communication occurs asynchronously. Similarly, the simulation should supply data to a port and indicate this to the hardware for processing (e.g. through a callback function).

Files can be thought of as a special type of device, and correspond to files stored on disk. These are referenced by a file *path*, and can be both written to and read from, although the primary purpose of files is to log data from the simulation. Typically, writing to file occurs at a specified *rate*. This type of logging is distinct from that of saving the entire state of the simulation (snapshotting or checkpointing) such that it may be resumed in the case of faults. To borrow some terminology from Charm++, state information from simulation that is transformed into a more serialized format to be written to file (or device) is *packing*, and data that is processed from a device to be used in simulation is *unpacking* [1].

2.2. Edge. Edges provide the method of information exchange between vertices (and other edges), or *endpoints* for brevity. Like a vertex, edges also contain information about *state*, *parameters*, *events*, and their equations. Unlike a vertex, however, a good portion of the state and events that an edge process is *external* to that edge. This is because the primary role of edges is to communicate state information that is local to the endpoints, whereas the internal state of an edge is used to determine *how* that information is communicated. The *connection* that an edge makes is between a *source* list and a *target* list, with each entry given an identifying name that may be referenced by the edge model. Although the most common case for a connection is a single source and a single target, the use of lists enables multiple branches for more complex connection modeling.

2.2.1. Edge Types. The most basic type of edge simply connects endpoints to endpoints, and is classified as a *direct* connection. Here, the set of sources and targets is taken directly from the relevant lists, with the most common case being a one-to-one connection. Although this type of connection can perform rudimentary handling of one-to-many, many-to-one, and many-to-many connections, it is limited in that the set of sources and targets is limited to precisely the length of their respective lists.

For communication between an endpoint and a subgraph containing vertex and edge sets, we introduce the connection types *broadcast* and *reduce*. As their name suggest, the broadcast connection type communicates information from a source endpoint list to a target subgraph list. What this specifies is that all of the endpoints in the target subgraph get connected to the source. Going the other way, from subgraphs onto endpoints, the connection performs a reduction. Again, the connection is resolved down to the vertex and edge sets of the subgraph at one end, the source this time. These broadcast and reduce connection types find use in collective operations on the graph, with top-down control being a particular use-case on one side, and collective analysis on the other. Edges that operate on subgraphs are also responsible for invoking structural dynamics.

A few special subgraphs that are made available to an endpoint is all of the edges that connect to it: incoming, outgoing, and by class. In this way, even if an edge is *direct*, the

endpoint may address all of its connections in a collective fashion. This is particularly useful for broadcasting a spike out to all of a neurons post-synaptic connections with a separate process for broadcasting a spike out to all of its pre-synaptic connections, and for reducing location information in the computation of nonlinear operations at a dendritic tree.

2.2.2. Augmentation by Edges. An important feature in specifying the construction of the graph is the ability of the edges to augment the state (and dynamics) of their connections. Because vertices (e.g. a neuron) do not typically require external state in their simulation, the specification of a vertex can more or less be considered 'self-contained'. However, when information is exchanged between vertices through an edge, additional state may be required to model the full set of state dynamics. Examples of state augmentation of a connection would be the introduction of intermediate computations, inputs into the vertex, or modifications of existing state dynamics to events. Instead of specifying the neuron model to accommodate the additional information needed during communication, the burden of specification is placed on the connecting edge.

Although a typical use-case for state augmentation may be that of information exchange between two vertices, we do not limit ourselves to this. An important scenario where it is important to provide augmentation of edge state dynamics is in reward modulation. Here, an event (e.g. the release of dopamine) acts to modify the behavior of existing edges rather than the vertices. Through the use of subgraphs, an interaction such as this may be handled through the association of a vertex to an edge set. As far as constructing the subgraph is concerned, whereas vertex only sets may be determined at the same level as the graph first defining those vertices, subgraphs containing a non-empty edge set must be constructed in a higher-order graph. This is because there is a dependency on the existence of the edges in the graph to construct the subgraph from.

2.2.3. Edge Connectivity. All communication can be framed as the sending and receiving of messages between endpoints. How closely linked the information exchanged between them determines the connectivity of the edge. Based on this, we may classify edges as either *local* or *remote*. The standard connection is *local*, where information between the endpoints such as state and events may be shared. For state information, this occurs at a prescribed rate, such as after a set amount of integration, either at fixed time intervals, change in the state's value, or both. The classification of a local connection as far as simulation is concerned, provides a preference of an edge to remain on the same computation node such that it may take advantage of any shared memory. For *remote* connections, the information exchange is restricted to event-based communication only, compared to local connections which may transmit state information. Unlike state, the exchange of events is expected to occur much more sporadically, at the occurrence of the events. In simulation, the preference for vertices to reside on the same computation node of remote edges is relaxed from that of local edges, with the aim of reducing the amount of inter-node communication as much as possible.

2.3. Graph. As mentioned in the overview, the purpose of the graph component is to define the vertex sets and determine the relevant edges (to be provided as an edge set). The structural information given in a graph is specified in terms of its *vertices* and *edges* at a basic level, where each element may be referenced by a given *id*. Additionally, a graph may define *subgraphs* and *associations* at an abstracted level for more compact specification. Although graphs are used to tell the simulator what gets simulated, the vertices and edges, the graph component does not contain its own state or parametrization, and don't evolve according to any set of state dynamics. Rather the 'state' of a graph is its structure, represented in terms of its vertex and edge sets. How this structure may change in the course of a simulation is

an important process to study and methods for specifying structural dynamics are given in sec. 2.3.5.

2.3.1. Vertex Definitions. In the construction of a graph, we first define the vertex sets, taken from the vertex component in sec. 2.1. Any parameters or state dynamics may be overwritten during this definition according to sec. 2.1.2. Vertices defined by a graph take *from* vertices (or graphs) defined previously, and are given a new *name* or alias to be used for reference. Furthermore, the graph may perform any *initialization* of state of the vertex that may or may not have already been specified. For vertices that specify other graphs, there is no additional initialization procedure (the expectation is that the vertex initialization has already been performed). When a vertex is defined in a graph, it may take on additional structural information such as *position* in space. This spatial information may then be used to provide any spatially dependent connectivity rules.

Although vertices may be defined and instantiated on a vertex-by-vertex basis, this becomes intractable for large neural networks. Rather, *populations* of vertices may be specified where a particular vertex model is *copied* or replicated a given number of times specified by the *order*. As with individual vertices, populations are also given a *name*. To differentiate between the vertices of a population, each member is given an *index*, which is unique on a per-population or per-subgraph basis. For vertices in a population, the initialization of state and position is performed on a per-vertex basis.

2.3.2. Edge Binding. Perhaps the most important function of the graph is to specify how information is exchanged between vertices and edges. This is performed by *binding* external states and events to their respective destinations. The first step of this process is to bind the endpoints in terms of source and target lists. For direct edges, this may be performed on single and populations of vertices. Here, all combinations of vertices at their respective endpoints are iterated over to determine if a connection should be made. This determination is computed according to a *ruleset* of equations that produces a boolean true/false value or probability $p \in [0, 1]$, where 0 (or false) implies no connectivity, and 1 (or true) implies certain connectivity. There may be multiple equations in a ruleset, each providing its own value. Treating each condition as independent from one another, these values are all multiplied to generate a final probability. Because edges may access the structural information of its endpoints, connection rules according to spatial locality may be readily generated. The other edge types, reduce or broadcast, must be performed by supplementing the source or target, respectively, list with subgraphs, where a population is a special type of subgraph that contains only a vertex set. After determining if a connection is created, any external state or events must be resolved, either from edge to endpoint, or vice-versa. In the case of a local edge, this resolution may also bind the state of one endpoint to the state of another endpoint through augmentation (sec. 2.2.2).

2.3.3. Subgraph. Subgraphs provide a level of abstraction on top of graphs, enabling the construction of more complex structure in a compact manner. Unlike a graph, a subgraph may not define its own vertex sets. Rather, subgraphs obtain their sets of vertices and edges from an existing graph or subgraph. Accordingly, the order of construction of subgraphs is important: the vertex or edge set that a subgraph is conditioned over must already be defined. This means that while a subgraph may be constructed from vertices at the same level as the graph that defined those vertices, a subgraph may not be constructed from any of the edges that would be determined by that graph. A subgraph containing a non-empty edge set must be constructed at a level above the construction of the graph it is conditioned over (i.e. in a graph of graphs). This provides a well structured containment order for processes like structural dynamics.

The general construction of a subgraph is performed in a set-theoretic manner. *From* an existing vertex and edge set, we generate a subgraph by performing an intersection with a number of specified *properties*. As an example, we may have $S = \{G \mid P_1, P_2, \ldots, P_n\}$, where S is the subgraph, G provides the vertex and edge set of graph G, and P_i for $i = 1, \ldots, n$ are the set of properties to condition over. This is similar to the evaluation of a ruleset for binding edges, and shared by systems such as Neurons to Algorithms and Connection-Set Algebra [8, 4]. The term population from before may be used to refer to a subgraph containing only vertices, and populations of vertices may be treated as its own subgraph. The spanning subgraph, or *span* of a (sub)graph is particularly useful in the construction process, and contains all the vertices that are defined by that (sub)graph. During the construction process, the span of a (sub)graph is grown according to each instantiation of a vertex. An *induced* subgraph may also be generated from a graph by conditioning over its span with a vertex set or population.

2.3.4. Association. Whereas vertices are connected by edges, subgraphs are associated by schemes. As an extension to connections that are specified by edges, association schemes specify more complex connections across the vertex and edge sets of subgraphs. This is more of an all-to-all connection as opposed to the *direct*, *reduce*, and *broadcast* edge types found in sec. 2.2.1. However, associations use the more basic edge types to perform the underlying connectivity, and there is never a case where a subgraph is directly connected to another subgraph. Just as with the abstraction of subgraphs, associations also do not contain any of its own state or parametrization. Rather, associations contain a number edges and their rulesets. Additionally, associations may contain (intermediate) subgraph definitions that permit more complex interconnectivity among the vertices of the source and target subgraphs. All the binding of state and augmentation of the endpoints is defined by an association just as the graph would define its edges (sec. 2.3.2).

2.3.5. Structural Dynamics. Although graphs and subgraphs don't contain their own state, they do provide important structural information that may be used in the simulation and study of structural dynamics. Information such as the *order* and *position* of the vertices of a subgraph may be referenced in the reduction operation of a collective or higher-order vertex and modified accordingly. Information about the *size* and *bindings* of the edge set may also be referenced and modified. The *class* and *index* of each vertex or edge is also available. In a sense, the subgraph may be thought of as a collection of pointers to the simulated objects, the vertices and edges. To some extent, this borrows from Charm++'s distinction between data- and work-units, where certain logical elements are responsible for the storage of state and structure and others are responsible mainly for the computation, respectively [1]. This is an idea that is heavily utilized in NAMD (NAnoscale Molecular Dynamics program) [7].

Events may modify vertex and edge sets of subgraphs, and contain the standard initialization procedures of state (like the vertex constructor in a graph). This, in turn, modifies the underlying graph, as well as any related subgraphs. Again, here is where the containment order of subgraphs is important in determining how they are affected by structural changes. When a vertex is added to a subgraph, that vertex is also added to any enclosing subgraphs (e.g. the span of a graph). Any subgraphs that are enclosed match the new vertex against its set of properties to determine if it should be added. For the addition of edges, a similar process is performed. Unlike vertices, however, edges may be added without actually updating any of the subgraphs of a graph. This is because it is possible for the subgraphs of a graph to be only populations. Rather, an edge generated at the same level of the graph simply adds to the adjacency lists of relevant endpoints. Higher-order graphs will see the edge in any subgraphs it is a member of, however (e.g. lower-order-graph.span.induced).

Structural events include the *birth* or *death* of a vertex (or population of vertices), and whether to *attach* or *detach* endpoints through an edge. These events occur as a result of network 'growing rules' that are generated by a higher-order vertex, but defined by the encapsulated subgraph. These definitions follow the same definitions of vertices and edges presented in sec. 2.1 and 2.2. Here, the source and targets of an edge are provided specific endpoint instances. The ability to *create* or *remove* a subgraph and to *acquire* or *forget* an association may also be useful. Presumably, these would come into play in the birth of a higher-order vertex. Because of the structural dynamics, any potential edge connections to an endpoint is preemptively augmented such that it will be ready when a new binding occurs.

Another important process to model is the maturation of a growing neuron, where the otherwise constant parameter values have not settled yet. To accommodate this, the vertex class for that neuron contains considerably more state than the mature version, where the state reflects the changing parameters. As the neuron matures (indicated by its *age*), it may change or *transform* to a different vertex class. Similar to initialization, the *new* vertex takes values from the *old* vertex (through bindings if not immediately apparent). Any values that are missing from the old vertex must be initialized, and any values not present in the new vertex are effectively dropped (such as the age).

REFERENCES

[1] B. Acun et al., *Parallel programming with migratable objects: Charm++ in practice*, in High Performance Computing, Netwroking, Storage, and Analysis, SC14: International Conference for, IEEE, November 2014, pp. 647–58.

[2] P. W. Anderson, *More is different*, Science, 177 (1972), pp. 393–6.

[3] S. Crook et al., *Creating, documenting and sharing network models*, Network, 23 (2012), pp. 131–49.

[4] M. Djurfeldt, A. P. Davison, and J. M. Eppler, *Efficient generation of connectivity in neuronal networks from simulator-independent descriptions*, Frontiers in Neuroinformatics, 8 (2014), pp. 1–11.

[5] E. Nordlie, M.-O. Gewaltig, and H. E. Plesser, *Towards reproducible descriptions of neuronal network models*, PLoS Computational Biology, 5 (2009), p. e1000456.

[6] B. Oksendal, *Stochastic Differential Equations*, Springer-Verlag, 5 ed., 2002.

[7] J. C. Phillips, R. Braun, W. Wang, J. Gumbart, E. Tajkhorshid, E. Villa, C. Chipot, R. Skeel, L. Kalé, and K. Schulten, *Scalable molecular dynamics with namd*, J. Computational Chemistry, 26 (2005), pp. 1781–1802.

[8] F. Rothganger, C. E. Warrender, D. Trumbo, and J. B. Aimone, *N2a: a computational tool for modeling from neurons to algorithms*, Frontiers in Neural Circuits, 8 (2014), pp. 1–12.

Software and High Performance Computing

The articles in this section discuss the implementation of high performance computing (HPC) and productivity software. Among the HPC reports, a common theme is the use of Kokkos. Kokkos is a programming model and C++ library enabling multithreaded C++ code to be performance portable across many-core architectures, such as conventional multicore CPUs, the Intel Many Integrated Core coprocessor (MIC), and graphical processing units (GPU).

Bookey, Demeshko, Rajamanickam, and Heroux describe their performance-portable reference implementation of the High Performance Conjugate Gradient Benchmark (HPCG) using Kokkos. Their report focuses on three parallel implementations of the triangular solve kernel that implements the symmetric Gauss-Seidel preconditioner: level scheduling, graph coloring, and an inexact Jacobi iteration.

Champsaur and Lofstead explore components to enable in-memory, flexible workflows for disparate scientific codes. They use the molecular dynamics code LAMMPS and the mini-app GTCP, a proxy for the particle-in-cell Tokamak simulator GTC.

Diamond and Devine integrate capabilities in Trilinos's Zoltan2 and RPI SCOREC's ParMA. ParMA performs diffusive load balancing directly on a SCOREC PUMI mesh. They provide an interface to ParMA in Zoltan2, and they implement a mesh adapter to make Zoltan2 available to SCOREC. Then they use these new capabilities to apply Zoltan2's hypergraph partitioning and ParMA's diffusive load balancing to two problems that use the PUMI mesh database.

Eberhardt and Hoemmen describe block Compressed Sparse Row (CSR) matrix-vector multiplication algorithms. They implement these using Kokkos and measure performance on conventional CPU, Intel MIC, and GPU architectures.

Eller and Edwards design, implement, and assess the performance of a thread scalable concurrent unordered map. This map is part of Kokkos's containers library.

Furst, Prokopenko, and Hu implement an adapter in MueLu for the NVIDIA AMGX algebraic multigrid GPU library and compare AMGX and MueLu performance on a single node.

Held and Bradley develop a correctness and performance assessment framework for a prototype multithreaded sparse triangular solver. The framework assembles a large test set of matrices from the University of Florida Sparse Matrix Collection, factorizes them, and writes data files as input to a test driver. Then it parses driver output to assemble results.

Kelley, Siefert, and Tuminaro implement a visualization tool to analyze the aggregation process in MueLu. It can be optionally run during the hierarchy setup and directly outputs parallel VTK files for viewing in ParaView.

Kumar and Hammond use the Structural Simulation Toolkit (SST) to model and analyze the performance of the compressible multiphase turbulence code CMT-nek. They focus on how the communication algorithms perform as the machine and problem sizes grow. They use SST Motifs to model the behavior of the network end points.

Lohrmann and Widener describe a new operator implemented in the EVPath communication middleware system. They discuss its application to a directory service for registering callback handlers and to SmartPointer, a LAMMPS data analysis program.

Morales, Littlewood, and Moore parallelize the Neohookean material model in ALbany's Laboratory for Computational Mechanics (LCM) using Kokkos. They measure speedup on a GPU relative to serial execution on a conventional CPU.

Munn and Moreland develop a mini-app focused on reference MPI and OpenMP implementations of the Marching Cubes algorithms. The Marching Cubes algorithm is applied

to image voxels to extract a triangulated isosurface. Their report provides background on the algorithm and then describes software design and data structure details of their implementations.

Raitses and Grant use the PowerAPI, a portable library abstracting access to vendor-specific power measurement tools, to profile MiniMD, a mini-app proxy to the molecular dynamics code LAMMPS. They find that MiniMD has a power profile similar to LAMMPS and so is a useful proxy for studying mechanisms to monitor and adjust power usage relevant to LAMMPS.

Staten, Carpenter, and Robinson describe a graphical user interface (GUI) to visualize unstructured triangular equation of state tables. A visualization tool can help with equation of state model development.

A.M. Bradley
M.L. Parks

December 18, 2015

PERFORMANCE PORTABLE HIGH PERFORMANCE CONJUGATE GRADIENT BENCHMARK

ZACHARY A. BOOKEY*, IRINA P. DEMESHKO†, SIVASANKARAN RAJAMANICKAM‡, AND MICHAEL A. HEROUX§

Abstract. The High Performance Conjugate Gradient Benchmark (HPCG) is an international project to create a more appropriate benchmark test for the world's largest computers. The current LINPACK benchmark, which is the standard for measuring the performance of the top 500 fastest computers in the world, is moving computers in a direction that is no longer beneficial to many important parallel applications. HPCG is designed to exercise computations and data access patterns more commonly found in applications. The reference version of HPCG exploitx only some parallelism available on existing supercomputers and the main focus of this work was to create a performance portable version of HPCG that gives reasonable performance on hybrid architectures.

1. Introduction. The High Performance Conjugate Gradient (HPCG)[5][4] is emerging as a complement to the High Performance Linpack (HPL) benchmark for ranking the top computing systems in the world. HPCG uses a preconditioned conjugate gradient to solve a system of equations, that executes both dense computations with high computational intensity and computations with low computational intensity such as sparse matrix-matrix multiplications.

The goal of our project was to create a performance portable version of HPCG that gives reasonable performance on all existing supercomputers. We choose Kokkos[2] library from Trilinos[3] as a tool to provide performance portability in HPCG code.

2. HPCG. HPCG is a new and upcoming benchmark test to rank the world's largest computers. On top of solving a large system of equations, HPCG also features a more irregular data access pattern so that data access affects results as well as matrix computations.

HPCG begins by creating a symmetric positive definite matrix and its corresponding multigrid to be used in the preconditioning phase. For the preconditioner it uses a Symmetric Gauss-Seidel forward sweep and back sweep to solve the lower and upper triangular matrices. For the actual solve of $Ax = b$, HPCG uses the conjugate gradient method after the preconditioning phase. HPCG runs in seven major phases.

1. **Problem Setup:** This is the beginning of HPCG and is where we construct the geometry that is used to generate the problem. HPCG generates a symmetric, positive definite, sparse matrix with up to 27 nonzero entries per row depending on the local location of the row.
2. **Validation Testing:** This portion of the program is to make sure any changes made produce valid results. Specifically it checks to make sure that both the unpreconditioned and preconditioned conjugate gradient converge in around 12 and 2 iterations respectively. It also makes sure that after performing both a sparse matrix vector multiplication and a symmetric Gauss-Seidel sweep that we preserve symmetry by using two randomly filled vectors and performing simple operations that should be zero due to the nature of our symmetric matrix A.
3. **Reference Sparse Matrix Vector Multiplication and Multigrid Timing:** This portion of the code times how long it takes to perform the reference versions of SPMV and Symmetric Gauss-Seidel.

*St. John's University, zabookey@csbsju.edu
†Sandia National Laboratories, ipdemes@sandia.gov
‡Sandia National Laboratories, srajama@sandia.gov
§Sandia National Laboratories, maherou@sandia.gov; St. John's University, mheroux@csbsju.edu

4. **Reference Conjugate Gradient Timing:** Here we run 50 iterations of the reference version of the conjugate gradient method and record the resulting residual. This residual must be attained by the optimized version of conjugate gradient no matter how many iterations are required.
5. **Optimized Conjugate Gradient Setup:** Runs one set of the optimized conjugate gradient and determines the number of iterations required to reach the residual found before. Then figures out how mamy times to reach the desired residual to fill in the requested benchmark time.
6. **Optimized Conjugate Gradient Timing:** Runs the optimized conjugate gradient the required amount of times. Records time for each timed section to report out later.
7. **Report Results:** Writes out log files for debugging and creates the .yaml file to display the results which can then be submitted if all the requirements are met.

HPCG gives you the option to run with MPI, OpenMP, both, or in serial. Running with MPI adds an extra dimension to the problem and requires processes to exchange values on their borders to perform. This results in a trade-off between more overhead and more parallelism.

3. Kokkos. As computer architectures differ in their features for best parallel performance it has become increasingly difficult to write code that will perform well across many different types of architectures. One solution to this problem is the C++ package, Kokkos. Kokkos acts as a wrapper around your code to allow you to specify at compile time where and how you want to run your application. Kokkos executes computational kernels in fine-grain data parallel within an `Execution space`. Currently Kokkos supports the following `Execution spaces`:

- `Serial`
- `PThreads`[6]
- `OpenMP`[8]
- `Cuda`[7]

Kokkos has two main features: `Kokkos::View` polymorphic Multidimensional Arrays and parallel dispatch. `Kokkos::View` is essentially a wrapper around an array of data that gives you the option to specify which `Execution space` you want to store the data on and allows you to choose what sort of memory access traits you wish this data to have. `Kokkos::View` also handle their own memory management via reference counting so that the view automatically deallocates itself when all of the variables that reference it go out of scope, thus making memory management much simpler across multiple devices.

There are three main parallel dispatch operations in Kokkos: `parallel_for`, `parallel_reduce`, and `parallel_scan`. All of these serve their own purpose and act as wrappers over how you would execute a section of code in parallel over the respective `Execution space`. For all of the data parallel executions kernels you initiate the kernel by passing in a functor that performs the desired parallel operation, such as from host to device.

`Parallel_for` is simply a generic for loop that will run all of the context of the loop in parallel. This works well for parallel kernels like vector addition.

`Parallel_reduce` is for simultaneously updating a single value, this function guarantees that you avoid race conditions with the updated values. `Parallel_reduce` works well for parallel kernels like finding the dot product of two vectors.

`Parallel_scan` is for taking a view and creating a running sum of values to replace the values of the view. Although `parallel_scan` is useful it was only really needed for setup phases in our HPCG.

One of the useful features of Kokkos is that parallel operations can be nested. This allows us to run up to three level parallelism inside a single kernel. This happens by creating a league of thread teams so that each team of threads has a specified number of threads. Then we can assign vector lanes to each thread that can run in parallel as well. This allows us to call a parallel kernel on the league and then another parallel kernel on the teams and finally a parallel kernel on the vector lanes of the thread. Depending on the type of problem nested parallelism can significantly improve performance of the Kokkos kernels, but, at the same time, is introduces some overheads that can be significant for the problems with no enough parallelism. We'll explore this further in HPCG later.

4. HPCG + Kokkos. The goal for our project was to create a version of HPCG that produces valid results across many architectures without sacrificing performance. We believe that Kokkos library is the best available tool to provide performance portability for the C++ code, we choose to re-factor HPCG to use it.

General strategy for Kokkos re-factoring includes:

- Replace custom data types with Kokkos multidimensional arrays;
- Replace the parallel loops with Kokkos parallel kernels;
- Code optimizations to improve usage of co-processors.

4.1. Replacement of custom data types with Kokkos multidimensional arrays. Restructuring the code involved a whole rewrite of HPCG to change how all of the structs stored their values. We replaced every array that would be used in a parallel kernel with an appropriate `Kokkos::View`. Once this was functional we had to go back to some of the compute algorithms and change how the data was accessed as to not try to access device data from the host or the other way around.

While restructuring we decided to change how our SparseMatrix stored the data and implemented it as a sort of overlying structure on top of a Kokkos CSRMatrix. This change required us to again go back and change how most of our computational kernels worked and created a noticeable increase in performance. At this point the code was functional across all of Kokkos execution spaces but took a severe performance hit while trying to run on Cuda.

4.2. Replacement of the parallel loops with Kokkos parallel kernels. Rewriting the parallel kernels involved replacing the parallel loops with the correct type of Kokkos parallel kernel. This part of re-factoring was heavily focused on converting the computation algorithms into functors and lambdas. Completing this task didn't affect portability at all and due to some Kokkos restrictions actually caused a slight reduction of performance in the function ComputeResidual. Other kernels would later have to be changed to accommodate the fact that data was stored on a device but was being run on the host.

An example of "nested loop to Kokkos" kernel conversion is presented in the Figure 4.1. Here we replace outer loop with the `Kokkos::parallel_for` and put internal part of the loop to the Kokkos kernel (see right part of the Figure 4.1).

Computing the preconditioner using a symmetric Gauss-Seidel was initially done in serial and thus moving to Kokkos required us to copy memory from the device to the host every time we ran it, which was the reason performance was lost. We tried implementing many different ways to perform a sweep of symmetric Gauss-Seidel in parallel to eliminate the need of copying data. We implemented an *inexact solve*, a *level solve*, and a *coloring algorithm*.

The *inexact solve*[1] works by performing a few triangular matrix solves. We split our

[1]We understand that the inexact solve violates the policies of the benchmark, but is included for com-

```
for (int i=0; i<localNumberOfRows; i++)
    double rowDot = 0.0;
    int diag_tmp;
    diag_tmp=A.values(diag(i));
    z_new(i)=r(i)/diag_tmp;
    z_new(i) += z_old(i);
    for(int k = A.graph.row_map(i); k <= diag(i); k++)
        rowDot += A.values(k) * z_old(A.graph.entries(k));
      z_new(i) -=rowDot/diag_tmp;
  }
}
```

```
class LowerTrisolve{
  public:
  local_matrix_type A;
  const_int_1d_type diag;
  const_double_1d_type r;
  double_1d_type z_new;
  double_1d_type z_old;

  LowerTrisolve(const local_matrix_type& A_,const const_int_1d_type& diag_,
          const const_double_1d_type& r_,
    double_1d_type& z_new_, const double_1d_type& z_old_):
    A(A_), diag(diag_), r(r_), z_new(z_new_), z_old(z_old_){
    Kokkos::deep_copy(z_old, z_new);
    }

  KOKKOS_INLINE_FUNCTION
  void operator()(const int & i) const{
    double rowDot = 0.0;
    double z_tmp;
    int diag_tmp;
    diag_tmp=A.values(diag(i));
    z_tmp=r(i)/diag_tmp;
    z_tmp += z_old(i);
    for(int k = A.graph.row_map(i); k <= diag(i); k++)
      rowDot += A.values(k) * z_old(A.graph.entries(k));
    z_tmp -=rowDot/diag_tmp;
    z_new(i)=z_tmp;
  }
};

...
Kokkos::parallel_for(localNumberOfRows, LowerTrisolve(A.localMatrix, A.
       matrixDiagonal, r.values, z, A.old));
..
```

Fig. 4.1: An example of "nested loop to Kokkos kernel" conversion.

matrix A into parts L, U, and D where L is the lower triangular matrix including the diagonal, U is the upper triangular matrix including the diagonal and D is the diagonal. The *inexact solve* is done by solving three different equations. Given our problem $Ax = b$, to simulate a symmetric Gauss-Seidel sweep we solve $Lz = b$ then $Dw = z$ and finally $Ux = w$. All of these solves are done using a parallel Jacobi solve. Although this version of symmetric Gauss-Seidel is run in parallel and avoid all device to host memory copies it still performs slowly since each Jacobi solve needs to be run a certain number of sweeps depending on problem density and results in more parallel kernels being launched than is ideal.

Our *level solve* algorithm splits our matrix A into parts L, U, and D where L is the lower triangular part of A that includes the diagonal, U is similar to L since it is just the upper triangular part of A that includes the diagonal. D is just the diagonal of A. For this algorithm we introduce another data structure, levels, to our SparseMatrix that stores all of the data needed for sorting the matrix. When we optimize the problem we find dependencies in solving for L and sort based on those dependencies in a way that a row will only be solved if all of its dependencies have been solved. We repeat this process for solving for A and store all of this data into levels. Now when we compute our Symmetric Gauss-Seidel we solve just like we do for the inexact solve except we start by solving for the rows in level 1 in parallel and then the rows in level 2 in parallel and repeat until we solve all of our levels. In the end we have a Symmetric Gauss-Seidel that has introduced a deal of parallelism and performs well compared to the original implementation. At the time of writing our *coloring algorithm* is not fully implemented. We are in the process of finishing it to provide it as part of HPCG+Kokkos implementation. The algorithm works by coloring our matrix A so that all rows with a certain color have no dependencies on one another. This way we can run a sweep of symmetric Gauss-Seidel in parallel over each color. While similar to the level solve, this method requires more iterations to converge due to the fact that although no two rows in the same color depend on each other it is likely that they depend on a row in a color that will be solved in a later iteration.

---

parison anyways.

5. Performance evaluation. We evaluate performance for our implementation of the HPCG code on our Shannon testbed cluster. Shannon has 32 nodes with 2 Intel Sandy Bridge CPUs and 2 Nvidia K40/K80 GPUs per node.

Comparing different variations of the preconditioner requires us to consider different problem sizes since some of our implementations will perform better on larger more sparse matrices than they will on smaller and denser matrices.

As you can see in Table 5.1 the standard version of symmetric Gauss-Seidel performs best since the level solve and the inexact solve only have limited parallelism. This is due to the fact that there aren't enough non-dependent rows to provide sufficient parallelism on each level and the inexact solve needs 18 iterations to get a solution for each triangular matrix that is close enough to the exact solution to maintain symmetry.

	Standard SYMGS	Level Solve	Inexact Jacobi Solve
Cuda (Shannon)	0.408776	0.0634977	0.091606
OpenMP (Shannon)	1.67202	0.233525	0.295145
Serial (Shannon)	1.46676	0.22152	0.269744

Table 5.1: GFLOPS Results for various SYMGS on Problem Size 16^3

As seen in Table 5.2 the level solve begins to become the optimal preconditioner. This is similar to the issue with size 16^3 where now our matrix is large and sparse enough that the level solve starts to see an increased amount of parallelism. The inexact method still lags behind since even though our matrix is less dense it still needs 12 Jacobi iterations to have a solution exact enough to pass the symmetry test.

	Standard SYMGS	Level Solve	Inexact Jacobi Solve
Cuda (Shannon)	0.64954	0.720434	0.560632
OpenMP (Shannon)	1.76462	2.42687	0.97734
Serial (Shannon)	1.45816	1.56839	0.485523

Table 5.2: GFLOPS Results for various SYMGS on Problem Size 64^3

Table 5.3 shows us still that level solve is our most optimal preconditioner for similar reasons as stated before. However it makes sense to note that the inexact solve is steadily increasing performance as we our problem size increases. With a problem size of 128^3 we only need 5 jacobi iterations to achieve a near exact solution to each triangular solve.

	Standard SYMGS	Level Solve	Inexact Jacobi Solve
Cuda (Shannon)	0.660526	2.1373	1.39231
OpenMP (Shannon)	1.82678	4.19081	1.84878
Serial (Shannon)	1.49575	2.18794	0.810011

Table 5.3: GFLOPS Results for various SYMGS on Problem Size 128^3

In Table 5.4 we tested our preconditioning algorithms on a problem size of 192^3. As expected our standard Gauss-Seidel performed at the same level as before and our level solve starts to slow down its performance increases. The inexact solve left us with some

abnormally high results that remained valid. For executing on OpenMP we witnessed results of 100+ GFLOPS and for Cuda we found high 90's. We are still investigating what may have caused these results.

	Standard SYMGS	Level Solve	Inexact Jacobi Solve
Cuda (Shannon)	0.659346	2.89654	???
OpenMP (Shannon)	1.842	4.858	???
Serial (Shannon)	1.5048	2.37896	???

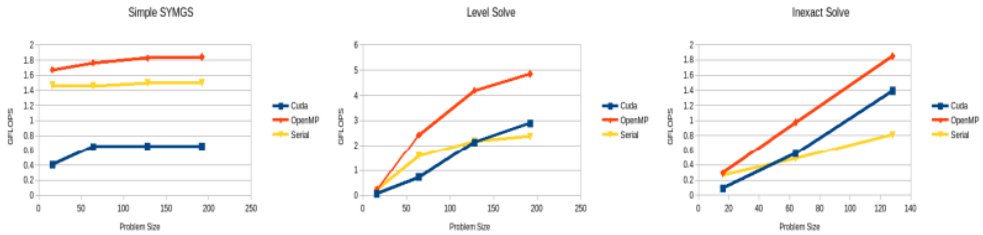
Table 5.4: GFLOPS Results for various SYMGS on Problem Size 192^3

Now we're going to look at how each preconditioning algorithm performs overall on each execution space.

As seen in Fig 5.1(a) the vanilla symmetric Gauss-Seidel is dominant on the non-cuda execution spaces. This has to do with the fact every time this method is called we had to run a lot of memory copies between the host and device. When we aren't using Cuda all of our memory is located in one space and thus the memory copying is avoided and we don't see a huge performance hindrance.

In Fig 5.1(b) the results between execution spaces while using the level solve preconditioning method vary quite a bit. First, notice that there is a trend with the results and it appears that our max performance received from this preconditioner starts to taper off. With this trend in mind it appears that OpenMP will achieve better performance using this preconditioner than Cuda. However this could be because we haven't done much cuda optimizations in terms of memory.

Looking at Fig 5.1(c) it seems we have a trend that increases our performance based on problem size. We are confident this has to do with the fact that when our problem size is larger our matrix is less dense and so we need less jacobi iterations to produce an answer exact enough to pass the symmetry tests. However we opt to leave out problem size 192^3 due to the abnormal results mentioned before.



((a)) Simple SYMGS ((b)) Level Solve ((c)) Inexact Solve

Fig. 5.1: Plots of GFLOPS over the Various SYMGS Algorithms

5.1. Optimizations. Aside from the symmetric Gauss-Seidel optimizations already made, we are looking to further these optimizations in the future by exploiting hierarchical parallelism. For example, in the *inexact solve* we can implement a three level parallelism technique that breaks our rows into chunks that a team of threads each gets assigned. Now our teams of threads work on our chunk while in parallel we perform the matrix vector

computation in parallel using the threads in the team. A near identical method can be applied to our *level solve algorithm* but since the level solve deals with fewer rows at a time we won't see as much of a performance increase in here. We're confident that these aren't the only places that can benefit from hierarchical parallelism and we plan to explore these options at a later time.

If we start by looking at our original LowerTrisolve kernel used in the inexact solve algorithm (see Figure 4.1) we see that there is a for loop inside of our parallel kernel that doesn't update any outer values and could benefit from being parallelized. If you are even more clever you can take advantage of the third level by assigning chunks to teams as described above. Thus taking full advantage of three level parallelism we are left with the following.

Below is a code snippet that demonstrates how we used Kokkos hierarchical parallelism in the lower trisolve kernel for the inexact solve version of the symmetric Gauss-Seidel.

```
class LowerTrisolve{
   public:
   local_matrix_type A;
   const_int_1d_type diag;
   const_double_1d_type r;
   double_1d_type z_new;
   double_1d_type z_old;
   int localNumberOfRows;
   int rpt = rows_per_team;

   LowerTrisolve(const local_matrix_type& A_,const const_int_1d_type&
       diag_, const const_double_1d_type& r_,
         double_1d_type& z_new_, const double_1d_type& z_old_, const int
             localNumberOfRows_):
         A(A_), diag(diag_), r(r_), z_new(z_new_), z_old(z_old_),
             localNumberOfRows(localNumberOfRows_){
         Kokkos::deep_copy(z_old, z_new);
         }

   KOKKOS_INLINE_FUNCTION
   void operator()(const team_member & thread) const{
      int row_indx=thread.league_rank()* rpt;
      Kokkos::parallel_for(Kokkos::TeamThreadRange(thread, row_indx,
          row_indx+rpt), [=] (int& irow){
         double rowDot = 0.0;
         double z_tmp;
         int diag_tmp;
         diag_tmp=A.values(diag(irow));
         z_tmp=r(irow)/diag_tmp;
         z_tmp += z_old(irow);
         const int k_start=A.graph.row_map(irow);
         const int k_end=diag(irow)+1;
         const int vector_range=k_end-k_start;
         Kokkos::parallel_reduce(Kokkos::ThreadVectorRange(thread,
             vector_range),
               KOKKOS_LAMBDA(const int& lk, double& lrowDot){
               const int k=k_start+lk;
               lrowDot += A.values(k) * z_old(A.graph.entries(k));
            }, rowDot);
         z_tmp -=rowDot/diag_tmp;
```

```
            z_new(irow)=z_tmp;
        });
    }
};

..
  const int team_size=localNumberOfRows/rows_per_team;
  const team_policy policy( team_size , team_policy::team_size_max(
      LowerTrisolve(A.localMatrix, A.matrixDiagonal, r.values, z, A.old,
      localNumberOfRows) ),vector_lenght);
  Kokkos::parallel_for(policy, LowerTrisolve(A.localMatrix, A.
      matrixDiagonal, r.values, z, A.old, localNumberOfRows));
```

Using the above implementation of nested parallelism in our inexact solve method for symmetric gauss-seidel, we see performance results as in Figures 5.2, and Tables 5.5 and 5.6. It is clear that Cuda receives a huge benefit from this as it allows us to finely tune how we want to parallelize this method. This nested implementation works as described above and for our figures it chooses chunks of size n for a problem of size n^3. This assures us that we actually work with every row in our matrix and gives us decent results.

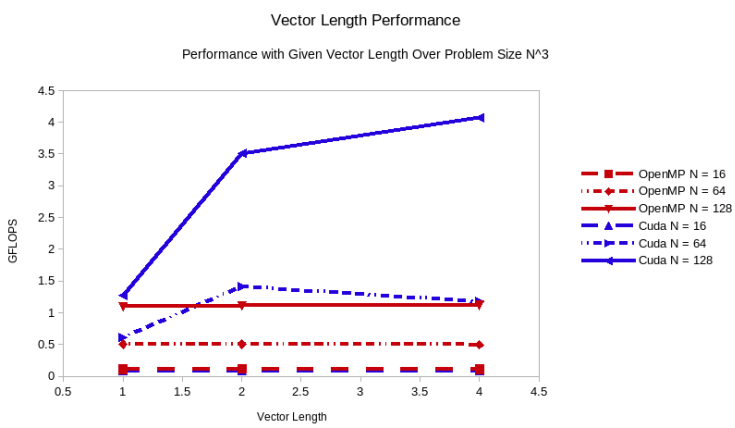


Fig. 5.2: Plot of GFLOPS for different vector lengths used in lowerTrisolve

Vector Levels:	1	2	4
$N = 16$	0.087516	0.0857736	0.0834751
$N = 64$	0.611932	1.41309	1.17325
$N = 128$	1.2713	3.51299	4.07047

Table 5.5: GFLOPS for Number of Levels and Problem Size N^3 Affect Cuda Performance

These results are highly preliminary and we are going to investigate performance for different combinations of the number of rows per team, vector length and problem size.

Vector Levels:	1	2	4
$N = 16$	0.110544	0.111938	0.111116
$N = 64$	0.50259	0.502229	0.495932
$N = 128$	1.09584	1.11385	1.11723

Table 5.6: GFLOPS for Number of Levels and Problem Size N^3 Affect OpenMP Performance

6. Conclusion. In the end we have worked towards creating a version of HPCG that works alongside the Kokkos package found in Trilinos. This version of HPCG will be a useful for being able to run the reference version of HPCG out of the box and not need to configure the code to be compatible with the specific machine being benchmarked.

Our work here is not completed but we have made great headway on this project and currently have code that produces similar results across all of the Kokkos execution spaces. In the future we plan to fix a few performance bottlenecks and utilize hierarchical parallelism to fully take advantage of Kokkos kernels.

We also plan to fix our coloring algorithm for the symmetric Gauss-Seidel so we can choose at compile time which algorithm to use for preconditioning. It will be interesting to compare performance between all of our preconditioning algorithms.

REFERENCES

[1] J. Dongarra and et al., *Top 500 supercomputer sites.* `http://www.top500.org`, 1999.
[2] H. C. Edwards, C. R. Trott, and D. Sunderland, *Kokkos: Enabling manycore performance portability through polymorphic memory access patterns*, Journal of Parallel and Distributed Computing, (2014).
[3] M. A. Heroux, R. A. Bartlett, V. E. Howle, R. J. Hoekstra, J. J. Hu, T. G. Kolda, R. B. Lehoucq, K. R. Long, R. P. Pawlowski, E. T. Phipps, A. G. Salinger, H. K. Thornquist, R. S. Tuminaro, J. M. Willenbring, A. Williams, and K. S. Stanley, *An overview of the trilinos project*, ACM Trans. Math. Softw., 31 (2005), pp. 397–423.
[4] M. A. Heroux and J. Dongarra, *Toward a new metric for ranking high performance computing systems*, tech. rep., Sandia National Laboratories, 2013.
[5] M. A. Heroux, J. Dongarra, and P. Luszczek, *Hpcg technical specification*, Tech. Rep. SAND2013-8752, Sandia National Laboratories, 2013.
[6] laise Barney, *Posix threads programming.* `https://computing.llnl.gov/tutorials/pthreads/`.
[7] NVIDIA, *Cuda programming guide version 3.0*, tech. rep., Nvidia Corporation, 2010.
[8] OpenMP Architecture Review Board, *OpenMP Application Program Interface*, 1023.
[9] Y. Saad, *Iterative Methods for Sparse Linear Systems*, Society for Industrial and Applied Mathematics, 2003.

INSIGHTS FOR THE DESIGN AND USE OF GENERIC SCIENTIFIC WORKFLOW COMPONENTS

ALEXIS P. CHAMPSAUR[†] AND GERALD F. LOFSTEAD, II[‡]

Abstract. In the context of scientific applications in the HPC space, real-time scientific workflows reduce the need for writing large amounts of raw scientific data to disk and for the multiple exchanges between analysis codes and disk often used in offline data analysis. However, even though many workflows use similar calculations, assembling them is a complex task and modifying them requires significant effort, since these are usually tightly linked codes designed for use with specific scientific applications.

As part of the Decaf project, we explore in this work the requirements of components that could be linked together to assemble a wide variety of workflows driven by a variety of scientific codes. We present insights in the design of such tools based on our implementation and use of 4 generic workflow components assembled into two worklows, each driven by a different scientific code having a large user base.

1. Introduction. As HPC platforms approach exascale, the scientific applications that run on them are allowing us to discover more information than ever before. However, scaling these codes and the analytical components that are usually used with them to levels theoretically obtainable on today's hardware presents significant challenges. Indeed, a majority of simulations that target HPC platforms are still designed to write their output directly to disk, with analysis performed *offline* (after the data has been written to disk), even if the size of the data of value in the end is only a small fraction of that output by the simulation. It is well-known that the I/O stages of these codes are bottlenecks in their performance [3] [9]. The multiple I/O routines between simulation, data analytics, and disk often used in this context take up valuable time and compute resources, and they limit the amount of information we can obtain from these applications. The development of parallel I/O methods such as MPI-IO [8] and netCDF [4] has helped alleviate this problem by speeding up I/O stages. However, these techniques in themselves do no address the discrepancy between the sizes of raw simulation output and final results of value.

More recently, there has been significant effort in defining frameworks and developing techniques that allow data processing codes to operate while the simulation is still producing data. This is termed *online analytics*. The Adaptable I/O System (ADIOS) [5] was designed in part to facilitate this by providing an abstraction of the data as a stream, rather than only as files that exist on disk or in memory. Flexpath, an implementation of this stream-based interface, allows simulation processes and various analytics processes to exchange data in real time, regardless of their numbers or how they are arranged in the system. In-situ methods [9] take advantage of locality by placing analytics on the same nodes as the processes that produce the data. Such tools allow the creation of real-time scientific workflows, where complex scientific computations are performed by linking together a number of concurrently executing components.

Still, while some tools exist for stream-based processing, workflows that use online analytics are still often designed *ad hoc*, with the same types of analytics often re-written for various workflows. These workflows are usually tightly coupled: each component is designed to receive data in exactly the format output by the previous component in the flow. This approach has inherent disadvantages: (1) the same types of data manipulation and analysis components are often re-written; (2) any desired modification to the workflow requires complex changes to the code; (3) real-time adjustments to the workflow are not possible. The Decaf Project [6], an ASCR data management project of which this work

[†]Georgia Institute of Technology, alexis.champsaur@gatech.edu
[‡]Sandia National Laboratories, gflofst@sandia.gov

is part of, aims to develop generic primitives that would allow scientists to piece together workflows in a simplified and flexible way. Its main goal is to discover ways in which to "loosen the grip of tight coupling" for the reasons just provided, and this is also the focus of this work. In the Decaf project, we wish in the long term to allow for the *automatic* creation of entire portions of workflows. Such "broadly applicable dataflows" [6] would be based on higher-level decisions made by scientists who wish to construct workflows. This type of functionality will require at its core a set of components that are adaptable to a variety of data properties, that use consistent semantics, and that do not sacrifice performance. How to approach the design of such components is what we explore in this research.

1.1. Contribution. In this work, we offer some insight in the design of generic data manipulation and analysis components from our implementation of two workflows. These workflows are driven by two different scientific codes, but they use some of the same components. Our key insights are these: (1) to allow for the greatest variety of workflows, data manipulation primitives and data analysis components should be packaged in similar ways — that is, regardless of their invidual complexity, the pieces that make up these workflows should export compatible interfaces as much as possible; (2) the ability to handle multi-dimensional data, along with the consistent labeling of dimensions and quantities as meta-data, allows for components that are highly adaptable and simple to use; (3) while different types of components understand varying levels of semantics, maintaining a high level of semantics (labeling quantities and dimensions as much as possible) early on and when passing through components that do not necessarily require all of these labels allows for the most functionality downstream; (4) because programming languages understand multi-dimensional data as being in a specific order in memory, there is a need for components that re-arrange data and re-label its dimensions without necessarily changing its size. Indeed, when data is contained in a database on disk, it is simple to gain a desired view of the data, for example by using SQL. However, in the middle of a real-time workflow, data must be presented to the components in a format that they expect and understand. By allowing workflow components to support any number of dimensions, by labeling these dimensions consistently, and by developing components that re-arrange and re-label data, we can do this in a generalizable fashion.

2. Design. We designed and implemented two real-time workflows based on scientific codes having large user bases: the LAMMPS Newtonian particle simulator [7], and GTCP, a proxy version of the particle-in-cell Tokamak simulator GTC [2]. While both of these workflows eventually turn the simulation data into **histograms** of certain quantities of interest, how they arrive at their final result varies significantly. Creating similar types of results, and this using some of the same components but in significantly different ways has allowed us to gain important insight into how best to design components that can be used in a wide variety of workflows.

In the next few paragraphs, we first describe the workflows from a general point of view, and then describe the individual components in greater detail.

2.1. Workflows. In the first workflow we implemented (Figure 2.1), LAMMPS outputs a number of quantities for each particle in the simulation at certain timestep intervals. This corresponds to two-dimensional data, and among the output quantities are the three-dimensional components of the particles' velocities. Data arrives from LAMMPS at the first component we designed, *Select*, which extracts these velocity components from the data output by the simulation. From *Select*, data is sent to *Magnitude*, another of our components, which computes the magnitudes of the velocities. In our current implementation, *Magnitude* outputs one-dimensional data (an array of the magnitudes it calculates) to the

final component, *Histogram*, which expects one-dimensional data as input. The end result of this workflow is a series of histograms of the total velocities of the particles. There is one histogram created at each timestep at which the simulation would normally dump its data to disk.

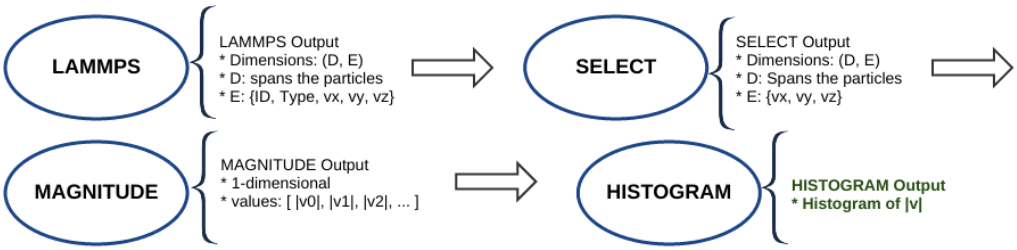


Fig. 2.1: LAMMPS Workflow

The second workflow (Figure 2.2) is driven by GTC, a code that simulates a toroidally confined plasma. The simulation splits the solid into toroidal slices, each made up of a number of grid points, and for each of these it outputs 7 properties of the plasma such as pressure and energy flux. The output of the simulation is therefore a three-dimensional array in which the indices represent: (a) toroidal rank (toroidal slice number), (b) grid point number, and (c) property number (flux, parallel pressure, etc.). In our workflow, data first arrives at an instance of *Select*, which extracts one quantity of interest out of the 7. This quantity is the "perpendicular pressure," or pressure of the plasma perpendicular to the flow in the grid point of interest. Even if it contains only perpendicular pressures, the output of *Select* is still three-dimensional, since this component maintains the original dimensions of its input. Because the *Histogram* component expects one-dimensional input, we first send the output of *Select* through two instances of our *Dim-Reduce* component, each of which eliminates a single dimension of the array without changing its total size. The final component, *Histogram*, outputs a histogram of the perpendicular pressures of all grid points at each timestep at which the simulation would normally output its data to disk.

2.2. Components. Even though we refer to the components as single entities, they are distributed codes that know how to split computation among the processes of which they are composed. They use ADIOS for both input and output, which allows them to consider the data as a stream. Flexpath, which implements the stream-based data exchange abstracted to the components through the ADIOS interface, is asynchronous, and allows for data exchange between any number of writers and readers. Therefore:

1. We can launch components of the workflow in any order: downstream components will wait for the availability of data from upstream components, and upstream components will buffer data up to a certain size until they are able to send it downstream. This also means that the decision as to which downstream components to use can be made after the upstream components have started running, allowing for real-time adjustments to the workflow based on results obtained upstream.
2. Even if the number of processes used for one component is different from that used for the previous one in the workflow, each component can split the data (and therefore the computation) evenly among its processes. We should mention, however, that due to the current implementation of Flexpath there is overhead data

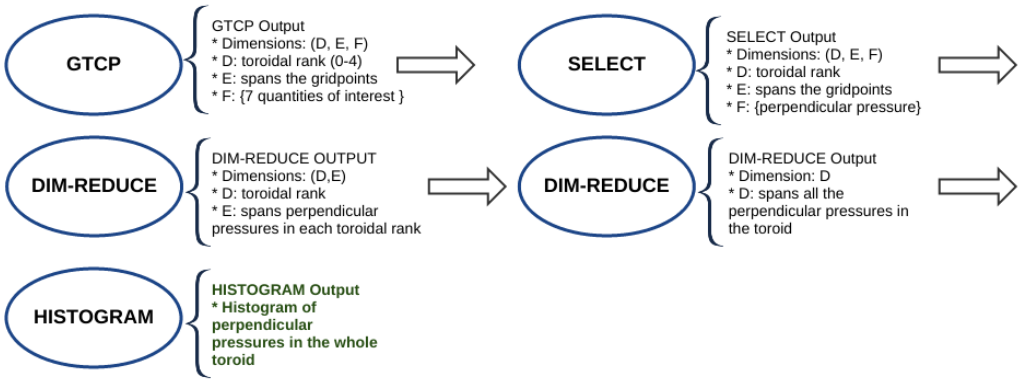


Fig. 2.2: GTCP Workflow

exchanged when different numbers of writers and readers are used. Even if reader R requests only a portion of writer W's data, the current implementation is such that W sends all of its data to R.

In addition, ADIOS and its transports (such as Flexpath) keep track of the dimensions of data and of the sizes of these dimensions. Therefore, when a component receives a multi-dimensional array, it can discover the dimensions of the data and their sizes as defined by the previous component in the workflow.

When using a component, one must specify the names of the input stream from which to read, the array in the input stream, the output stream to which to write, and the name of the array to use in the output stream. Referring to streams and arrays using names allows users to easily chain together these components into potentially complex workflows. Certain components require more information from the user. For example, *Select* must know from which dimension to select the quantities of interest.

2.2.1. Select. Given an input array with any number of dimensions, *Select* extracts certain indices from one of the dimensions and outputs an array with the same number of dimensions, but with the dimension of interest having a smaller size. The output data of this operator therefore has a smaller overall size than its input data. In order to select the quantities of interest, the component uses a header which must be passed by the previous component in the workflow. The header is a list of strings that name the quantities in the dimension of interest. This allows easy selection of quantities at runtime when *Select* is launched. For example, in the LAMMPS workflow, the simulation outputs the ID, Type, V_x, V_y, and V_z of each particle, where V_x, V_y, and V_z are the components of the velocity of the particle. *Select* discards the ID and Type of the particle, building a new array consisting only of the velocity components of each particle. The user (or a higher-level dataflow assembler) must pass to this component the index of the dimension from which to select.

2.2.2. Magnitude. In our current implementation, magnitude expects a two-dimensional array as input, where one dimension spans the data points at each time step (particles in the case of LAMMPS and grid points in the case of GTC), and the other dimension spans any number of components of the same quantity, for example the three-dimensional components of velocity in the LAMMPS workflow. *Magnitude* calculates the magnitudes of these quantities from their components and outputs a one-dimensional array of new values. Which

dimension is which in the input array is specified by the user at runtime.

2.2.3. Dim-Reduce. As discussed previously, workflow components must receive data in a format that they expect. For example, *Histogram* expects one-dimensional data. *Dim-Reduce* is a data manipulation component that removes one dimension from its input array, "absorbing" it into another dimension without modifying the total size of the data. The other dimensions are left unchanged. This component can work with an input array having any number of dimensions. The output is an array with one dimension removed, and with another dimension that has been re-defined. We discuss this operator and the need for it in more detail in Section 4. When using this component, the user must specify which dimension to eliminate and which to grow.

2.2.4. Histogram. The processes that make up the *Histogram* component partition among themselves a one-dimensional array of data. They communicate to discover the global mininum and maximum values in the array, create a number of bins between these two extremes, and then communicate again to count the number of values in the globally partitioned array that fall in each bin. The number of bins to use must be passed to the component when it is launched.

In our current implementation, one of the processes of *Histogram* writes the output to a file on disk. We chose this approach because this component is generally used as an endpoint in the workflow, and because the output of this component is generally small and can be easily written by a single process. However, as we discuss later, letting this component output its data in the same way as the other components (as an ADIOS stream), and instead writing to disk when needed using a component specifically designed for this purpose would provide greater flexibility.

2.3. Modification of the Simulation Code. In order to use some of the same components in both workflows, we had to slightly modify the output stages of the scientific codes driving them. Because in both workflows, the first component to receive the simulation data is *Select*, each simulation has to write a header of its quantities in the dimension to be selected from. Also, normally LAMMPS packs its two-dimensional output into a single array. We modified this so as to let it write a two-dimensional array, which better describes the output data and allows downstream components to better understand it. Both simulations had to be modified to enable the use of ADIOS for output.

3. Evaluation.

3.1. LAMMPS workflow. We ran the LAMMPS workflow on the Rhea cluster [1] at Oak Ridge National Laboratory, using a total of 100 nodes for the simulation and workflow components alltogether. Each Rhea node is capable of running 16 concurrent processes. The table below shows the distribution of processes in the workflow.

Number of Particles	**Total Procs**	**LAMMPS Procs**	**Select Procs**	**Magnitude Procs**	**Histogram Procs**
20,243,885	1600	800	288	256	256

Table 3.1: LAMMPS Worflow Execution, over 6 simulation write steps

We used data size as the primary factor in determining an appropriate number of processes to use for the workflow components. The total output size of the simulation with this configuration is approximately 0.95 GB per time step. With the above configuration, this lets each *Select* process handle about 1.2 MB. While we statically determine the *proportion* of processes to use for simulation and workflow components, a bash script appropriately

scales, distributes, and launches these processes as a single job for the PBS scheduler. In the configuration we used, any node handles only one type of component, and entire nodes are allocated to specific components. While in-situ computation is often advantageous [9], it is not the preoccupation of this work and we decided to avoid it alltogether.

The time between the start of the simulation and end of the last histogram write was 73.57 s. In contrast, the time we measured for the simulation with the same configuration to run and write all of its output directly to disk was 37.23 s. To truly determine the advantage of such a workflow, we would have to time a similar *offline* workflow. However, being able to obtain valuable results from real-time analysis in about twice as long as it takes the simulation to run and write raw data directly to disk is promising in itself.

3.2. GTCP Workflow. We tested the GTCP workflow on the same cluster. This time, we used a total of 128 nodes, distributed in the manner shown in table 3.2.

Grid Points	**Total Procs**	**GTCP Procs**	**Select Procs**	**Dim-Reduce1 Procs**	**Dim-Reduce2 Procs**	**Histogram Procs**
73,770,780	2048	768	512	256	256	256

Table 3.2: GTCP Worflow Execution, over 5 simulation write steps

As an example of the data produced by this workflow, consider the histogram created at time step 15:

Bin low	**Bin high**	**Grid points**
-0.023674	-0.019983	8
-0.019983	-0.016292	36
-0.016292	-0.012602	154
-0.012602	-0.008911	1188
-0.008911	-0.005220	15450
-0.005220	-0.001530	959515
-0.001530	0.002161	72410815
0.002161	0.005852	373301
0.005852	0.009542	9333
0.009542	0.013233	843
0.013233	0.016923	107
0.016923	0.020614	20
0.020614	0.024305	7
0.024305	0.027995	1
0.027995	0.031686	2

Table 3.3: Histogram of GTCP Perpendicular Velocities, Timestep 15

Obtaining such histograms from two very different simulations but using some of the same components required adjusting in small steps the way the components worked so as to make them fit in both workflows. This allowed us to gain a number of insights the design of such tools.

4. Insights.

4.1. Multi-Dimensional Data Support. Many scientific codes serialize their output, effectively packing multi-dimensional data into a single dimension. However, this technique offers little information on the data to downstream components in a workflow. In the case of our LAMMPS workflow, for example, if we kept the simulation output one-dimensional, downstream components would have to be specifically designed to read data in this format, and such components could not be easily made to work with serialized data having other formats. In our example, a separate component would have to be designed to work with the output of GTCP.

In order to use the same *Select* component downstream from both LAMMPS and GTC, we had to (a) modify the output of LAMMPS to let it output data in a format where the dimensions are clearly defined, and (b) we had to design *Select* in such a way that it understand input data having any number of dimensions. Indeed, even with the modification to LAMMPS, the simulation's output is two-dimensional, whereas the output of GTCP is three-dimensional.

In general, it is advantageous to (1) design components that can operate on multi-dimensional data as much as possible, and (2) format the ouput data of scientific applications as having well-defined dimensions. Emphasizing the support for multi-dimensional data in the design of workflow components allows for maximum compatibility between the interfaces of components by providing a consistent way to refer to the data. With *Select* supporting any number of dimensions, the user can simply indicate at runtime which dimension to select from. Similarly, in *Dim-Reduce*, any dimension can "absorb" any other dimension, as long as these dimensions are correctly specified by the user. Such functionality increases the generic quality of these components and simplifies their use. In our implementation, *Magnitude* expects two-dimensional data, but allowing this component to support any number of dimensions would be both feasible and advantageous.

Still, while multi-dimensional data support provides a consistent way to refer to the data, not all components should be designed so as to work with ***any number*** of dimensions. For example, we found it advantageous to design *Histogram* to work with only one dimension. Creating a histogram is a calculation that makes sense for one-dimensional data, and supporting a higher number of dimensions would add unnecessary complexity this component. If the data has more dimensions than are expected, or if only certain indices in one dimension hold values of interest, we can use the *Dim-Reduce* and *Select* components to properly format the data for *Histogram*, as we do in the GTCP workflow.

4.2. Semantics. When data is organized under clearly defined dimensions, labeling these dimensions as the data goes through each component lets downstream subscribers refer to dimensions using their names. Because the data is potentially re-sized and re-arranged in the course of a workflow, it is useful to maintain such semantics as much as possible. However, the absence of labels should not block the workflow execution. We did not label dimensions in the components and workflows we implemented, rather refering to them by number, but this optional functionality can be easily added and the advantage of doing so in the development of more complex workflows is clear.

In both workflows, we do however label the ***quantities*** in one of the dimensions so that the *Select* component can extract some of these quantities using their names. These names are concatenated into a "header string" using a well-known character as a separator. Because *Select* is used as the first analytical component in both workflows, we are only concerned with the existence of this header in the simulations and in the *Select* components. We write this header in the output of the simulations and read it when the data arrives at *Select*. Using *Select* further downstream poses a problem, however. In our current implementation, knowning which quantities to select requires such a header, and ensuring the existence of this

header at any point downstream would require labeling all quantities at every component, which would be highly impractical. As a solution, we suggest providing a choice to the user between using a header and using index numbers to select quantities. Then, if the header exists, the quantities' names can be used, and if *Select* needs to be used downstream and there is no header describing the quantities, the user can provide the indices of the quantities to select. We see here that labels should be used as much as possible, but that they should also be kept *optional*.

4.3. Dimension Reduction (*Dim-Reduce*). In the context of scientific workflows, there are some situations where it is desirable to eliminate one of the dimensions of a multi-dimensional array without losing any of the values stored in the data set, and without changing the meaning of some of the other dimensions. For example, in its output, GTCP keeps track of the toroidal slice that produces the data of interest by using a dimension that spans the "Toroidal rank" of grid points. In our workflow, we wish to create histograms encompassing all grid points in the toroid, thereby eliminating the concept of "Toroidal rank" and instead growing the dimension that spans the number of grid points.

Programming languages represent multi-dimensional arrays in a specific order in memory, so we cannot simply keep the data ordered as it is, change the number of dimensions and their sizes, and assume that the new dimensions correctly reference the data. In fact, here we demonstrate that even though *Dim-Reduce* does not modify the size of the data, it potentially requires a rearrangement of the data in memory, so that the data is correctly represented by the new dimensions. We also show that a simple calculation can be used to obtain new indices in the dimension that grows.

4.3.1. Dim-Reduce: Description. Because the motivation for this operation is based on how people understand data, rather than how it is represented internally, it is best explained through an example. Say that we have a solid divided into 6 sections. In each section, there are 3 particles for each of 4 charge types and 2 colors. We have a 4-dimensional array holding the speeds of all particles, for each value of these attributes (section, particle number, charge type, color). The dimensions of the array are $N_0 \times N_1 \times N_2 \times N_3$, where:

- $N_0 = 6$ is the size of the *section* dimension D_0
- $N_1 = 3$ is the size of the *particle number* dimension D_1
- $N_2 = 4$ is the size of the *charge type* dimension D_2
- $N_3 = 2$ is the size of the *color* dimension D_3.

Now, say that for the purpose of performing some type of analysis on the dataset, we wish to eliminate the concept of *color* of a particle and compensate by changing the concept of *particle number*, so that D_1 spans all particles in a particular section and of a particular charge type, regardless of color. We still wish, however, to keep track of the original section where the particle is located and of its original charge type. The desired result of this operation is that:

- The concept of *color* of a particle disappears.
- The concept of *particle number* loses its original meaning and takes on a new one. Its dimension now spans all particles in a particular section and of a particular charge type, regardless of color.
- The concepts of *section* and *charge* keep their original meanings, and the sizes of their dimensions are unchanged.

We can say that D_1, the *particle number* dimension, ***absorbs*** D_3, the *color* dimension. The array resulting from this operation has dimensions $N'_0 \times N'_1 \times N'_2$, where:

- $N'_0 = 6$ is the size of the *section* dimension D'_0
- $N'_1 = N_1 \times N_3 = 6$ is the new size of the *particle number* dimension D'_1
- $N'_2 = 4$ is the size of the *charge* dimension D'_2.

We call this operation *dimension reduction*. Even though it does not change the size of the data set, we can show that it potentially changes the ordering of data in memory.

4.3.2. Dim-Reduce: Re-Ordering of Data. In the original array, say we are interested in the speeds of the particles in section number 3, and having charge type 1. This corresponds to all of the particles with indices $(3, p, 1, c)$, where p is any particle number and c is any charge type. Assuming row-major order, the one-dimensional indices of these quantities of interest are:

- 1-D indices of $(3, p, 1, c)$: $74, 75, 82, 83, 90, 91$.

After the dim-reduce operation described above, even though the concepts of particle number and color have changed, we would like the indices in the dimensions D'_0 and D'_2 to keep their original meanings. The operation leaves us with a $6 \times 6 \times 4$ array, in which the 1-D indices of the quantities of interest, that is, the speeds of the particles of charge type 1 in section number 3, are:

- 1-D indices of $(3, p', 1)$: $73, 77, 81, 85, 89, 93$.

In this case, we see that the one-dimensional indices of the same values before and after the dim-reduce operation all differ. Therefore, in order to satisfy the desired qualities of the dim-reduce operation, we potentially have to re-order the data in its one-dimensional representation.

4.3.3. Dim-Reduce: Calculation. Using the same example, say (s, p, ch, cl) are the indices of a value in the original array, and (s', p', ch') are the indices of the same value in the new array (after the *dim-reduce* operation).

Because the new dimension D'_1 holds new meaning for the particles, we have some freedom in what its indices represent, as long as they range from 0 to $N_1 \times N3$ in increments of 1.

If we define the dim-reduce operation as:

- $s' = s$
- $p' = p + (cl \times N_1)$
- $ch' = ch$,

then the above requirements are met. This is simple to generalize to any number of dimensions.

4.3.4. Dim-Reduce: Conclusions.

- One dimension is eliminated, another dimension takes on a new meaning, and all other dimensions keep their original meanings.
- This is useful in scientific workflows.
- It potentially requires a re-ordering of data in its 1-D representation.
- We can use a simple arithmetic operation to create new indices in the dimension that grows.

4.4. End Components. In its current implementation, *Histogram* outputs its data to a file on disk. This design decision was made because histograms are often end-results in workflows and because they are small. A more flexible design would be to let *Histogram* output its data in the same way as *Select*, *Magnitude*, and *Dim-Reduce*. Indeed, if we give all data manipulation and analysis components the possibility of sending their output to downstream components, we do not have to designate some of these analysis tools to be workflow endpoints. Instead, designing a specific *End Component* whose sole purpose it to write to disk would provide greater flexibility in the arrangement of the other components.

5. Conclusions and Future Work. There exists today a wide variety of scientific workflows in the HPC space. These complex scientific codes often make use of the same

types of analytical procedures, but they are too often designed from scratch for each application that they work with. This leads to tighlty coupled workflows, which are difficult to maintain and modify. Instead, building workflows from existing components is easier and more flexible. Stream-based, generic workflow components should be designed so as to allow for the greatest variety in their arrangement and for a maximum number of downstream subscribers. Designing components with the ability to handle data having any number of dimensions provides a very useful way to link them together. Maintaining a high level of semantics upstream, for example by labeling dimensions and certain quantities inside of these dimensions, gives a good understanding of the data to downstream components. There is a need for components that organize the data in a format that downstream components can understand. And, designing specific disk writer components removes the need to temporarily modify analytics components to let them also act as disk writers.

There are a number of improvements we can make to our current implementation to have at our disposal more robust and flexible workflow components. As mentioned earlier, reading and writing dimension labels at each step in the workflow provides more information to downstream components and presents a clear advantage. The ADIOS interface includes the ability to send output to multiple destinations, by having several "write groups." We wish to explore the possibility of a *Fork Component* that would use this functionality of ADIOS to allow the creation of branched workflows.

The components we have developed in this research cover only a small portion of the procedures that computational scientists need in the development of their workflows. Eventually, we wish to allow for the development of a large collection of generic workflow components. We can take steps in this direction by building on our existing components. For example, *Magnitude* performs a relatively simple operation on multi-dimensional data, where one dimension spans a number of quantities involved in each instance of the operation. This model can fit any number of operations involving a repeating, fixed number of quantities, and it can even be made to work with a formula specified by the user at runtime. This opens the door to a large family of generic components.

Finally, while we have kept performance in mind in the development of these components, performance optimization is not yet the focus of this research. In the design of any *generic* tool however, the question of performance inevitably arises. Indeed, designing tools that are not meant to operate on a specific format of input data can easily impact performance. For example, *Dim-Reduce* performs the same amount of computation whether it re-arranges data or not. In the long run, optimizing these components will involve detecting such situations where they can avoid performing unnecessary iterations and data manipulation.

A. Initial Workflow Scaling Tests. To obtain a preliminary idea of the scaling potential of the workflows, we ran three configurations of the LAMMPS workflow and two configurations of the GTCP workflow on a 96-core testbed cluster at Georgia Tech. We tried to keep the data size per processor (both for simulation processors and component processors) roughly the same through all configurations in each workflow, and measured the time between the point at which the first component (*Select*) attached to the simulation's output stream and the point at which the last component (*Histogram*) finished writing its output. The numbers of processes used for the various components were chosen based mainly on the sizes of the data that they handled.

Particles	**Total Procs**	**LAMMPS Procs**	**Select Procs**	**Magnit. Procs**	**Histo. Procs**	**Time (s)**
846,301	48	23	10	10	5	56.51
1,281,601	72	35	15	15	7	41.96
1,713,101	96	46	20	20	10	64.30

Table A.1: LAMMPS Worflow Scaling

To exhibit ideal weak scaling, all of the above times would be equal. Below are the results for the GTCP workflow.

Grid Points	**Total Procs**	**GTCP Procs**	**Select Procs**	**Dim-Reduce (1+2) Procs**	**Histo. Procs**	**Time (s)**
1,027,570	22	10	6	4	2	36.74
4,103,134	96	44	25	18	9	76.93

Table A.2: GTCP Worflow Scaling

REFERENCES

[1] *Oak Ridge Leadership Computing Facility.* `https://www.olcf.ornl.gov/computing-resources/rhea/`, 2015. [Online; accessed 15-July-2015].

[2] S. Ethier, W. Tang, and Z. Lin, *Gyrokinetic particle-in-cell simulations of plasma microturbulence on advanced computing platforms*, in Journal of Physics: Conference Series, vol. 16, IOP Publishing, 2005, p. 1.

[3] S. Lakshminarasimhan, N. Shah, S. Ethier, S.-H. Ku, C.-S. Chang, S. Klasky, R. Latham, R. Ross, and N. F. Samatova, *Isabela for effective in situ compression of scientific data*, Concurrency and Computation: Practice and Experience, 25 (2013), pp. 524–540.

[4] J. Li, W.-k. Liao, A. Choudhary, R. Ross, R. Thakur, W. Gropp, R. Latham, A. Siegel, B. Gallagher, and M. Zingale, *Parallel netcdf: A high-performance scientific I/O interface*, in Supercomputing, 2003 ACM/IEEE Conference, IEEE, 2003, pp. 39–39.

[5] Q. Liu, J. Logan, Y. Tian, H. Abbasi, N. Podhorszki, J. Y. Choi, S. Klasky, R. Tchoua, J. Lofstead, R. Oldfield, et al., *Hello ADIOS: the challenges and lessons of developing leadership class I/O frameworks*, Concurrency and Computation: Practice and Experience, 26 (2014), pp. 1453–1473.

[6] T. Peterka, *Decaf Project Repository.* `https://bitbucket.org/tpeterka1/decaf`, 2015. [Online; accessed 1-July-2015].

[7] S. Plimpton, R. Pollock, and M. Stevens, *Particle-mesh Ewald and rRESPA for parallel molecular dynamics simulations.*, in PPSC, Citeseer, 1997.

[8] R. Thakur, W. Gropp, and E. Lusk, *On implementing MPI-IO portably and with high performance*, in Proceedings of the sixth workshop on I/O in parallel and distributed systems, ACM, 1999, pp. 23–32.

[9] F. Zheng, H. Yu, C. Hantas, M. Wolf, G. Eisenhauer, K. Schwan, H. Abbasi, and S. Klasky, *Goldrush: Resource efficient in situ scientific data analytics using fine-grained interference aware execution*, in Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, ACM, 2013, p. 78.

HYPERGRAPH PARTITIONING WITH LOCAL REFINEMENT FOR IMPROVING THE PERFORMANCE OF FINITE ELEMENT METHODS ON DISTRIBUTED UNSTRUCTURED MESHES

GERRETT F. DIAMOND* AND KAREN D. DEVINE†

Abstract. We consider partitioning algorithms to be used on unstructured meshes to improve metrics that are important measurements of the performance of finite element methods. These metrics include entity imbalance, the number of copied boundary entities, and total number of ghost elements across all part. Reducing these metrics and balancing them across the processors affect computational time and communication time as well as memory requirements per part. We propose the use of hypergraph partitioning followed by local refinement by diffusive load balancing to improve these metrics and the corresponding finite element methods' performance.

1. Introduction. Unstructured mesh partitioning and load balancing has been researched in great detail. Between diffusive methods [12, 18, 26, 30, 31], graph and hypergraph methods [4, 5, 6, 7, 8, 13, 15, 16, 19, 20, 21, 29], and multilevel schemes [3, 16, 19], the problem of equally distributing the mesh elements across processes has many algorithms to achieve near-optimal balance. However, many distributed applications that are run on meshes have overheads that are not directly affected by the element load imbalance. These applications require algorithms and schemes that balance other metrics as well as the element imbalance to get optimal performance and scalability. These constraints on runtimes and memory can lead to applications being impractical or impossible to run at large scales. These applications require new and more aware algorithms that partition especially for the specific metrics and needs of the application.

In this paper we describe two applications that require balancing beyond elements in Section 2. Section 3 lists previous work and some tools that are used and built on for this work. In Section 4 we detail the implementation of the work. Section 5 and 6 detail the experiment design and results of the proposed methods. Section 7 talks about future work and Section 8 concludes our study.

2. Motivation. The performance of several finite element methods (FEMs) is proportional to the imbalance of mesh entities as well as the number of copies that exist due to the partitioning of the mesh. In some methods, ghost entities are also a concern with respect to performance and memory. To optimally balance for these FEMs, partitioners are needed that include knowledge of copies and ghosts as well as entity counts. Here we describe two FEMs, one that depends on the total number of vertices, PHASTA, and one that includes layers of ghost elements on each part, MPAS-Ocean.

2.1. PHASTA. PHASTA [10, 23] is a massively parallel finite element flow solver. The computational load of PHASTA depends on the number of elements on each part in the equation formulation stage; the system solution stage depends on the total number of vertices. PHASTA uses element-based meshes which means each element is uniquely assigned to a part, but the vertices are copied when they are shared across more than one part. Thus the system solution stage and communication for it grows based on the number of shared vertices.

2.2. MPAS-Ocean. The Model for Prediction Across Scales (MPAS) is a modeling framework used for earth-system simulations. The MPAS framework is built on top of unstructured Voronoi meshes at large scales with regions of higher resolution. We specifically

*Sandia National Laboratories and Rensselaer Polytechnic Institute, diamog@rpi.edu
†Sandia National Laboratories, kddevin@sandia.gov

target the ocean modeling framework, MPAS-Ocean [24]. The MPAS-Ocean meshes consist of mostly regular hexagonal elements and require several layers of ghost elements per part for use in simulations. The total workload per part is proportional to the number of owned elements plus the number of ghosted elements. For large scale simulations on these meshes large ghost counts and imbalance of the ghosts across the parts can be a bottleneck in terms of communication costs and memory constraints as well as have an uneven workload. Sarje et al. [25] provide an example image of the workload imbalance by part in one MPAS-Ocean mesh seen in Figure 2.1. The coloring is done by process. Red processes correspond to more workload on that part and blue parts are less. Sarje et al. report that in this example the process with the highest workload has about three times as much work as the lowest process.

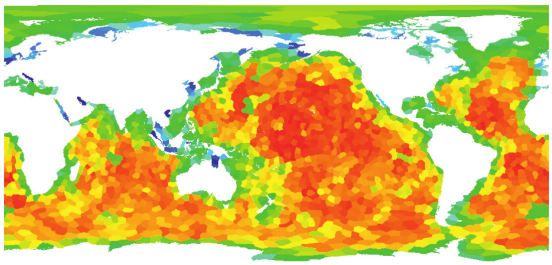


Fig. 2.1: Per-part visualization from [25] of the workload in an execution of MPAS-Ocean using a non-ghost-aware graph partitioning. Red parts have high work load and blue parts have low load.

Sarje et al. [25] use a ghost-aware hypergraph model to partition for the MPAS-Ocean problem. In their model they construct the k distant ghost adjacencies by computing the kth power of a matrix that represents the graph of the second adjacencies in the mesh. Then they iteratively partition using the hypergraph by assigning weights based on a cost predictive model of the computation and communication costs that depends on ghost cells. Sarje et al. also include ocean depth information per cell as the cell weights. In the paper, they show that the iterative hypergraph approach reduces the overall application time by up to 50% and also increases the scalability of the application.

3. Related Work.

3.1. Zoltan/Zoltan2. The Zoltan toolkit [2] provides abstract interfaces to a broad range of parallel algorithms that work on problems such as partitioning, ordering, and coloring. Zoltan uses callback functions to allow users to provide information on the data structures to be operated on by Zoltan's algorithms.

A new package, Zoltan2 [1], is in development as part of Trilinos [17] that uses C++ features to provide the functionality of Zoltan as well as expand the capabilities with more algorithms and interfaces. Zoltan2 also offers a more usable interface and the ability to construct adapters rather than callbacks to provide general information to the parallel partitioners, ordering methods and coloring algorithms.

3.2. PUMI and ParMA. The original goal of this work was to create an interface to SCOREC's Parallel Unstructured Mesh Infrastructure (PUMI) and Partitioning using Mesh Adjacencies (ParMA) [28] for use in Zoltan2. PUMI's main components are a parallel communication control, PCU, a geometric model interface, GMI, and a mesh interface, APF.

These packages are the basic framework behind ParMA and are necessary to use ParMA. ParMA is a collection of algorithms that use mesh adjacencies to perform diffusive load balancing directly on the mesh [28]. This is done iteratively, improving the target metric in each iteration. The main goal of ParMA is to balance the overall workload by diffusing high per-part workloads to neighboring parts and getting a better value of the target criteria. ParMA uses the complete mesh representation provided from PUMI and operates directly on the mesh instead of constructing a graph/hypergraph on top of the mesh.

3.3. Hypergraph Models. One way to solve the mesh entity copy problem is to build a hypergraph representing the mesh and partition with respect to hypergraph edge cut methods. Chevalier et al. [9] and Fortmeier et al. [14] describe the a hypergraph construction where each mesh element is treated as a vertex of the hypergraph and hyperedges are built on the mesh vertices. The hyperedge for a mesh vertex has a connection or pin to each element adjacent to the mesh vertex. Figure 3.1 shows an example hypergraph constructed on a small triangular mesh. The mesh is outlined in gray, the hypergraph vertices are in blue on each of the mesh elements and the yellow squares are the hyperedges on each mesh vertex. The pins of the hypergraph are presented as black lines. Fortmeier et al. describe in detail how this hypergraph construction models the communication by the hyperedge cuts as each hyperedge that is cut into more than one part represents a mesh vertex that is copied. A hyperedge that is cut multiple times creates multiple copies for that mesh vertex. Therefore Fortmeier's construction exactly models the communication problem that exists in the mesh entity copies problem.

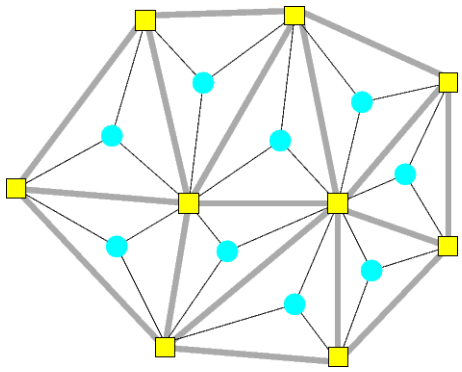


Fig. 3.1: A construction of the hypergraph on top of a mesh. The gray lines detail the triangular mesh, the blue circles are the hypergraph vertices and the yellow squares are the hyperedges. The black lines show the connectivity of the hypergraph.

Devine et al. [13] and Çatalyürek et al. [7] describe an implementation of the hypergraph construction with an exact model of communication. Their work is given in terms of matrices but is also applicable to unstructured meshes. Their parallel hypergraph partitioner, PHG, is available through the Zoltan toolkit and is the hypergraph partitioner that this work uses.

4. Implementation. Our proposed methods were implemented within Zoltan2. For this work, three new structures were added to Zoltan2: an algorithm, an adapter, and a model. The algorithm provides the interface from Zoltan2 to ParMA's load balancing algorithms. The adapter sets up use of Zoltan2 for the SCOREC mesh interface, APF. The new model is for hypergraphs to be built from a Zoltan2 adapter.

To use any of the SCOREC packages and code within Zoltan2, one must first pull in the SCOREC repository. This can be done by running the command: *"git clone https://github.com/SCOREC/core.git SCOREC"*. It is important to run this command in the root of the Trilinos repository and the directory must be named SCOREC. Then to enable SCOREC one must add the Trilinos_ENABLE_SCOREC:BOOL=ON flag to the cmake configuration. It is also possible to enable specific portions of SCOREC; for example, Trilinos_ENABLE_SCORECpcu:BOOL=ON enables the communication package of SCOREC. The full list of packages are gmi, pcu, apf, mds, parma, apf_zoltan, ma, and apf_stk which can be individually enabled like the previous example by prepending the package name with SCOREC. The SCOREC enable flag will enable all of the SCOREC packages.

4.1. The ParMA Algorithm. The first step in our work was to develop an interface to ParMA such that users of Zoltan2 can use the diffusive load balancing algorithms that exist there. The algorithm expects a MeshAdapter that includes either region or face data and adjacency to vertices. With this information, the algorithm constructs an APF mesh. Then, one of the balancing algorithms in ParMA is run on the APF mesh. The Zoltan2 Partitioning Solution is built from the new assignments that ParMA has assigned.

To build the APF mesh as an intermediate mesh representation, the mesh adapter must provide information about mesh vertices and mesh elements (either two dimensional or three dimensional). To fully construct the mesh, the adapter must provide first adjacencies from the elements to vertices. Also the elements need to have topological information defined as Zoltan2::EntityTopologyTypes. APF only supports the TRIANGLE, QUADRILATERAL, TETRAHEDRON, HEXAHEDRON, PRISM, and PYRAMID topologies. Coordinates on the vertices are also required for use in ParMA's centroid balancer.

There are seven ParMA balancers available through the Zoltan2 interface. The first two are a vertex balancer and an element balancer which attempt to decrease the imbalance of vertices and elements respectively through local diffusive migration of elements. The third and fourth balancers, vertex/element and vertex/edge/element, balance the listed entity types in order from left to right such that as the list is traversed and a given type is being balanced, the balance of preceding types is preserved. The fifth balancer is a centroid diffuser that balances elements but the elements that are migrated across parts are chosen based on distance from the part centroids. This balancer make more rounded parts while balancing the elements. The last two are specialized balancers, one that incorporates ghosts and one that targets part shape. The ghost balancer counts the number of ghosts each part has off each boundary that are within a certain number of layers and balances the number of ghosts plus the number of owned entities. The shape balancer unlike the other balancers may increase the imbalance in order to balance the part boundary length. The target of the shape balancer is to iteratively increase the size of short part boundaries to get a better average across the parts. Each balancer has a tune-able imbalance tolerance. For the first six balancers, the tolerance defines the target imbalance that ParMA will try to reach. These balancers will exit when either the tolerance is reached or a local minimum is detected that could not be improved over several steps. For the shape balancer the imbalance tolerance is the maximum allowable imbalance in the number of elements. Once over this tolerance, the shape balancer will terminate. The shape balancer will also terminate when the minimum part boundary length is greater than 70 percent of the average part boundary length.

The ParMA algorithms are accessible through a Zoltan2 Partitioning Problem by setting the parameter *algorithm* to *parma*. The user can choose which ParMA balancer to use by setting the *parma_method* parameter of a parameter sublist named *parma_parameters*. The seven ParMA balancers are accessed by the values *Vertex, Element, VtxElm, VtxEdgeElm, Centroid, Ghost*, and *Shape*. The default method is the vertex/element balancer. The

imbalance tolerance is set by the *imbalance_tolerance* parameter. The user can also set the degree of weight that will be sent on each iteration of ParMA by setting the parameter *step_size* of the parma_parameters sublist. A lower value makes ParMA send less weight per iteration; the default is 0.5. For the ghost balancer the depth of the ghosting and the through entity can be set using the *ghost_layers* and *ghost_bridge* parameters of the sublist. The number of layers defaults to 3. The bridge is the dimension of the entity (0 for vertex, 1 for edge, etc.) that defines second adjacencies in the mesh; it defaults to (element dimension-1).

4.2. APF Mesh Adapter. To follow up on the interface from Zoltan2 to ParMA, we wanted to lay the framework for SCOREC to use the various mesh partitioners of Zoltan2. For this idea, we implemented an APF Mesh Adapter that would convert the APF mesh database into the format used throughout Zoltan2. There are two key portions of the APF Mesh Adapter: the construction, and application of the partitioning solution.

The construction of the APF Mesh Adapter requires five sets of values to be defined. For the APF mesh we define all five sets on each dimension of the mesh to provide a complete description of the mesh. The first set defined is the ID numbers, both local and global. This was defined on the APF mesh using the Numbering and GlobalNumbering structures that assign unique numbers to each entity. The second set is the topologies of the entities. This requires a conversion from the apf::Mesh::Type to the Zoltan2::EntityTopologyType. The third set is the coordinates. For the vertices these are defined by the point location. For the other entities the coordinate information is computed by calculating the linear centroid of the entity. All coordinates of an APF mesh are always given in three dimensions. The fourth and final sets are the first and second adjacency information which are computed at the same time. First adjacencies are gathered locally per part using the apf::getAdjacency to get adjacencies from each dimension to every other dimension. Second adjacency is computed globally in a series of steps. The first step uses apf::getAdjacency on each of the first adjacent entities back to the original dimension. This gets all local second adjacency. To get global second adjacencies, a communication phase is used to share the data across copies. For non-element entities the communication is across all parts that the entity has residence on. For elements each element sends its own ID across any first adjacent entity to the resident parts of that adjacent entity. The receiving parts then add element IDs to the elements next to the copied adjacency.

The *applyPartitionSolution* function, which migrates mesh data to a new partition, is defined in two ways. The first is for elements. In this case the partitioning solution provides new locations for each element. Thus each element is migrated to its new destination and lower dimension entities are migrated and copied as needed. For a non-element entity partitioning, we have to convert the partitioning solution's part IDs to a partition of the elements. To do this we traverse each element and look at its downward adjacency to the primary entity. The element is assigned to the part receiving the highest number of its downward adjacent entities.

We also implemented two ways to assign weights to the mesh entities using a *setWeights* function. The first is a Zoltan2 style where the user provides an array and stride along with the mesh entity type so that entity e_i is assigned weight array[i*stride]. The second method is a more native way for users of APF. In this method the user defines an apf::MeshTag and assigns a weight to each entity of the dimension. The user then provides the apf::Mesh and apf::MeshTag along with the mesh entity type to the *setWeights* function. Any entity in that dimension that was not given a value is assumed to have a weight of one. We also assume that the each value in the tag is a weight so the size of the tag is the number of weights assigned to the entity. One can also assign multiple weights by calling the function

several times with different tags or with the array and stride method.

In early testing, we found that reading the entire apf::Mesh and storing each dimension was a large memory concern. To alleviate this memory constraint we took two approaches. The first is to not always compute the second adjacencies. This is both a runtime and memory concern as it requires a communication step as well as the storage for all second adjacencies. To make the mesh adapter compute second adjacencies a boolean flag is passed to the constructor. Our second approach is to not build every dimension of the mesh. In most cases Zoltan2 only needs the primary and adjacency types for the partitioning algorithms. So the constructor defaults to only storing information on these two types which are passed in to the constructor. To allow the user to build other types there is an optional integer argument that defines which dimensions to build other than the primary and secondary types. This integer should range from 0 to 2^{d+1}, where d is the dimension of the mesh. The integer represents a series of binary flags where the dth bit corresponds to whether or not the adapter will store the entities of dimension d. For example, passing in $9 = 1001_2$ will store the regions and vertices which are required by the Zoltan2 ParMA algorithm.

4.3. The New Zoltan2 Hypergraph Model. To run the hypergraph partitioners of Zoltan on the APF mesh we developed a hypergraph model as well as callbacks that work on the model to supply Zoltan with the data needed. This data includes the descriptions of hypergraph vertices, hyperedges, and pins. Two hypergraph model constructions were implemented on meshes for use in this work.

The first is the traditional hypergraph model used in hypergraph partitioning on meshes described by Fortmeier et al. [14]. In this model, mesh elements are represented by hypergraph vertices and the mesh vertices are the hyperedges. Then a pin goes from each hyperedge to any hypergraph vertex that represents a first adjacency from the mesh vertex to mesh element. Cutting a hyperedge is equivalent to making a copy of the corresponding mesh vertex. Therefore reducing the edge cut should decrease the number of copies and decrease the cost of communication in the finite element method. Thus the hypergraph partitioning targets reducing the imbalance of the elements as well as reduce the number of vertices copied. For our implementation of the hypergraph model we define the hypergraph vertices on the primary entities and hyperedges on the adjacency entities. This allows the above construction as well as any combination of entity dimensions to also create a valid hypergraph.

The second construction was one to target the number of ghost entities per part. In this model we represented the hypergraph vertices and hyperedges as the primary entities in the mesh. Pins were then added for any primary entity that was within k layers of second adjacent entities where k is just the number of ghost layers to count. These pins are added for on process and off process adjacencies. Rather than compute the pins via a BFS or DFS starting at each hypervertex, Sarje et al. [25] suggest a method using the adjacency matrix based on the graph model of the mesh. In this construction an adjacency matrix is built from the second adjacencies of the mesh between primary entities through the second adjacency type. This matrix defines the first layer of ghosts in the mesh. Subsequent layers of ghosts can be found by the k-th power of the adjacency matrix. All k layers of ghosting can be found by aggregating each power of the adjacency matrix. The size of the hyperedges in this model represent the total number of ghosts needed for each entity. Cutting a hyperedge results in more communication and copying of entities for the finite element method. Therefore the hypergraph partitioning will target decreasing the imbalance as well as the number of ghosts per part.

5. Experiments. All experiments were run on Red Sky, a capacity cluster at Sandia National Laboratories. Each node in Red Sky consists of a 2.93 GHz dual socket/quad

core Nehalem X5570 processor. Our experiments presented in this paper are run with up to 512 processes. For the PHASTA problem, we partition on two meshes: an abdominal aorta aneurysm (AAA) mesh [32] and a mesh of a tunnel with a bump midway through (deleryBump) [11]. Figure 5.1 shows the geometry of the two meshes. On the left is the AAA mesh and the right is the deleryBump. The AAA mesh has 2.1 million elements and the deleryBump contains 8.3 million. For comparison we perform runs on both meshes using the implemented hypergraph partitioner as well as Scotch graph partitioning [22]. For further analysis we run ParMA diffusive load balancing on both partitions using the VtxElm balancer.

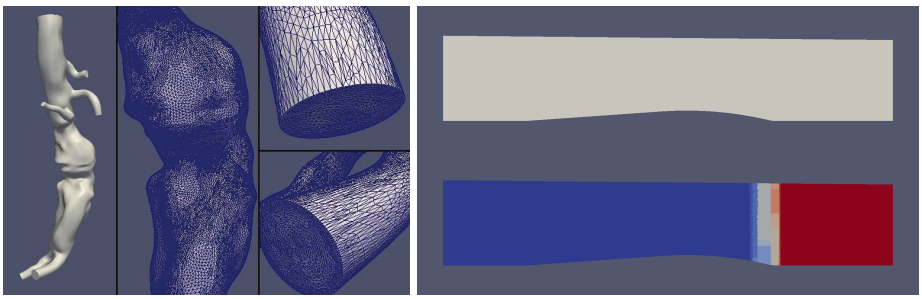


Fig. 5.1: (Left)An abdominal aorta aneurysm mesh [32]. (Right) a tunnel with a bump midway through. Bottom shows a 16-way partition of the mesh around the refined region.

For the MPAS-Ocean problem we use two ocean meshes of different resolution. The first has a resolution of 15km in the North Atlantic and 75km elsewhere. The other mesh has a uniform resolution of 15km across the mesh. Since the original mesh is mostly hexagonal and the APF interface does not support this type, we construct a triangular mesh where each vertex in the new mesh represents an element from the original. Figure 5.2 shows a small example of the hexagonal mesh and the mesh constructed for use by APF in red.

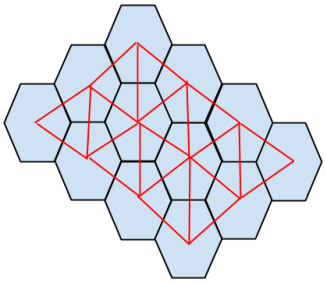


Fig. 5.2: The original hexagonal mesh and the corresponding APF mesh (in red)

The resulting North Atlantic mesh has 500 thousand elements and the 15km mesh has 3.7 million elements. We run our new ghost-aware hypergraph model where faces are the primary entity and edges are the adjacency entity type on these newly constructed meshes. For comparison we run similar constructions using a non ghost-aware hypergraph and Scotch graph partitioning. We count the hypergraph's off process ghosting which counts ghosts by

vertex, the part's ghosting which is the number of unique ghosts off a processes boundary, and the sum of a part's ghosts plus the owned. The meshes are partitioned using 16 to 128 processes on the North Atlantic mesh and 32 to 128 on the 15km mesh. We ran the experiments with three layers of ghosts.

6. Results. Figure 6.1 shows the results for the AAA mesh. The per-part maximum imbalance target for these runs is 1.05. It is known [32] and seen in these results that graph and hypergraph partitioning tend to have high imbalance on the vertices when partitioning elements. For this imbalance we use ParMA to lower the vertex imbalance with minor impact to the element imbalance. For the total number of vertices, the other metric for PHASTA, we see hypergraph partitioning gets much lower amounts than Scotch. Also running ParMA does not have a major impact on the number of vertices and for Scotch it decreases the number of vertices. The additional time required to refine partitions with ParMA is small for both hypergraph and Scotch partitions.

The results from running on the deleryBump mesh are in Figure 6.2. In this case, we see a larger imbalance of the vertices due to graph/hypergraph partitioning. Once again ParMA is able to lower this imbalance down to the target imbalance, 1.2. However, there is up to an 11% increase in the element imbalance after running ParMA on the hypergraph partition. Just as in the AAA partitions, hypergraph partitioning has significantly less vertices, but this time when we run ParMA there is a much more significant increase in the number of vertices. In the 512 part run, running ParMA on the hypergraph partition results in a higher number of vertices than running ParMA on the graph partition. We attribute this more significant increase in vertices to the increased vertex imbalance that ParMA is trying to reduce. ParMA partitioning time is greater for this mesh than for the AAA mesh; for this mesh and scale, ParMA takes 50% to 90% of the total partitioning time for Scotch partitioning with ParMA.

Due to its richer model, hypergraph partitioning is usually more expensive than graph partitioning. For these experiments, however, the hypergraph partitioning proved to be acceptably competitive. Our initial experiments showed that hypergraph partitioning was taking up to 60 times longer than Scotch. We identified a bottleneck in the implementation of the Hypergraph Model constructor that led to this unreasonable overhead. When primary entities may be copied on many processes, the Hypergraph Model includes an extra step that defines a unique "owner" or responsible process for each primary entity. The initial implementation of this step relied on many global Allreduce operations (one per primary entity). An improved implementation uses Tpetra's *Map* objects and *createOneToOneMap* function to more efficiently define owners for copied entities. Moreover, for all experiments in this paper, the primary entity type (element) is not copied in the APF mesh. We added a MeshAdapter method allowing users to specify which entities are not copied in their data, thus allowing us to skip this step when it isn't needed, drastically reducing the construction time for the Hypergraph Model. As a result, for the AAA mesh on 256 processes, Scotch partitioning took 16.1 seconds, while hypergraph partitioning took 20.4 seconds (1.3 times longer). For the deleryBump mesh on 512 processes, Scotch took 46 seconds; hypergraph partitioning took 103 seconds (2.2 times longer). In both cases, the performance of the hypergraph partitioner is sufficient to be used as an alternative to graph partitioning.

Figure 6.3 and Figure 6.4 compare the three partitioning approaches we used on the North Atlantic mesh and 15km mesh, respectively. In both meshes we see that traditional hypergraph partitioning does not work well in terms of ghost counts. In all of the runs there is a significantly greater number of ghosts compared to both the graph partitions and the ghost-aware hypergraph partitions. Between graph partitioning and the ghost-aware partitioning we see in all but one of the runs the ghost-aware has slightly fewer ghosts. The

reduction in ghosts using ghost-aware partitioning ranges from 0.1% to 5% compared with Scotch partitioning. In regards to timing, we see the ghost-aware hypergraph partitioning is fairly competitive with the unique-owner optimizations turned on. In the North Atlantic mesh, we see that both hypergraph models scale better than the graph partitioner; at 128 processes, the hypergraph and ghost-aware partitioners run faster than the graph partitioner. We see in all three algorithms that the imbalance of ghosts is quite high, between 1.4 and 2.1. The graph partitioning tends to have lower imbalance in the ghosts. This imbalance may lead to inefficiency in the application due to imbalanced communication costs. On the other hand we see that the ghost plus owned element imbalances are right around their target. For the two hypergraph models they target an imbalance of 1.1 which the ghost-aware model is at or below and the regular hypergraph fluctuates around. The graph partitions try to reach a lower imbalance when possible so we see a much lower imbalance in those cases.

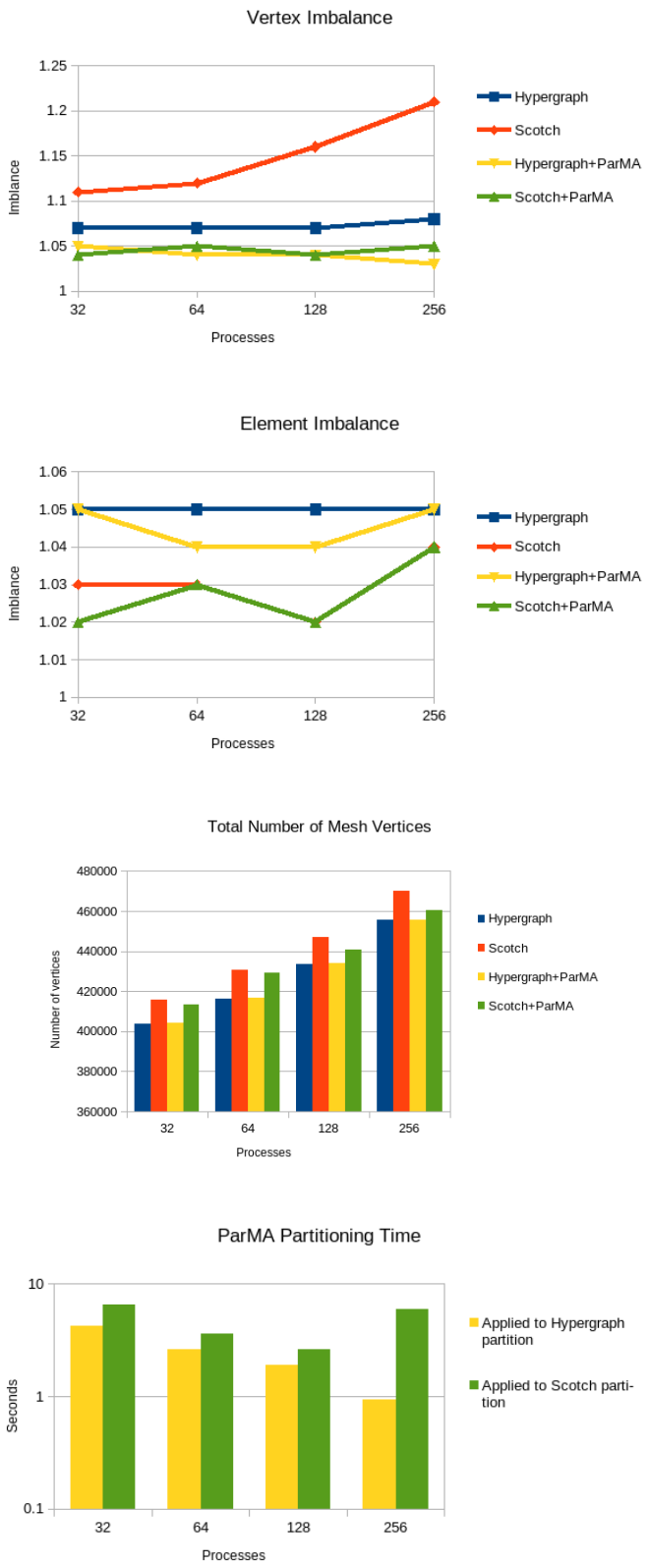


Fig. 6.1: The results from partitioning the AAA mesh. (Top) The vertex imbalance after partitioning. (Top Middle) The element imbalance after partitioning. (Bottom Middle) The total number of vertices in the mesh. (Bottom) The time in seconds for ParMA partitioning applied to hypergraph and Scotch partitions.

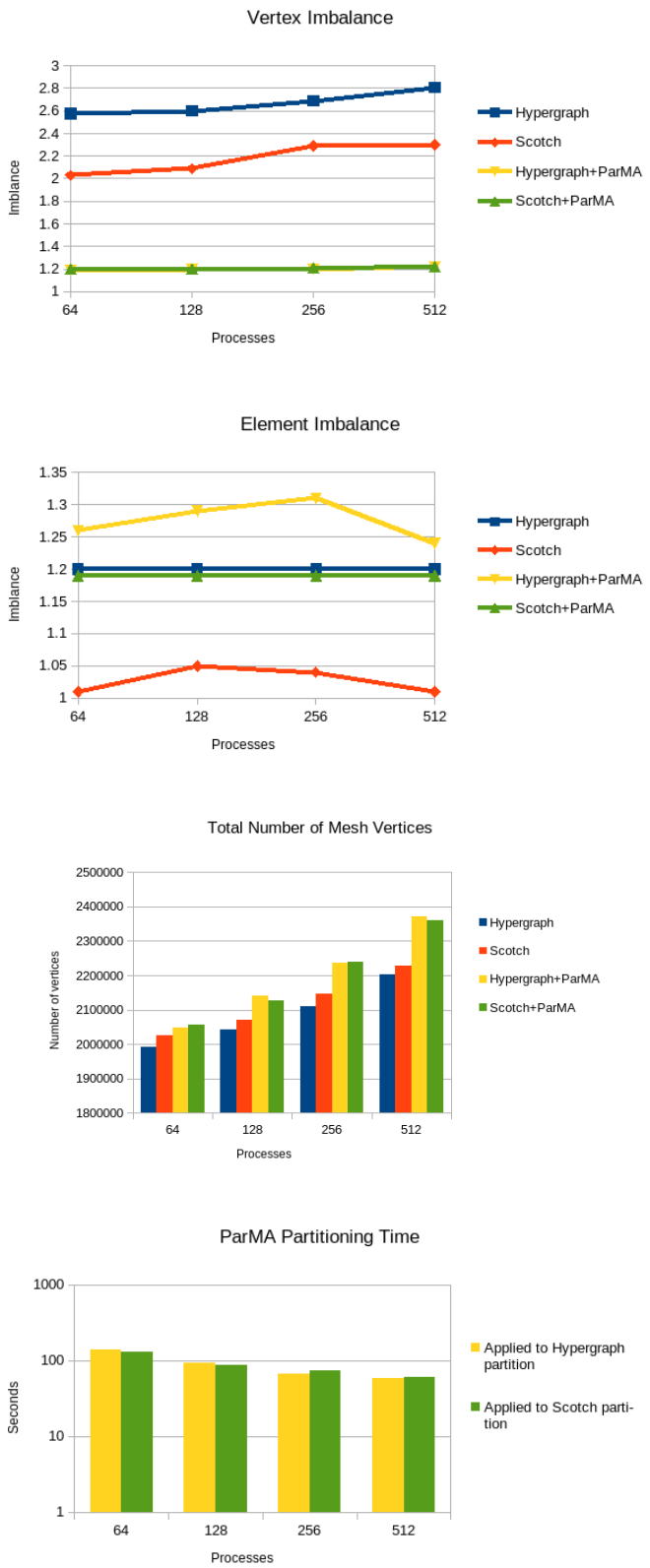


Fig. 6.2: The results from partitioning the deleryBump mesh. (Top) The vertex imbalance after partitioning. (Top Middle) The element imbalance after partitioning. (Bottom Middle) The total number of vertices in the mesh. (Bottom) The time in seconds for ParMA partitioning applied to hypergraph and Scotch partitions.

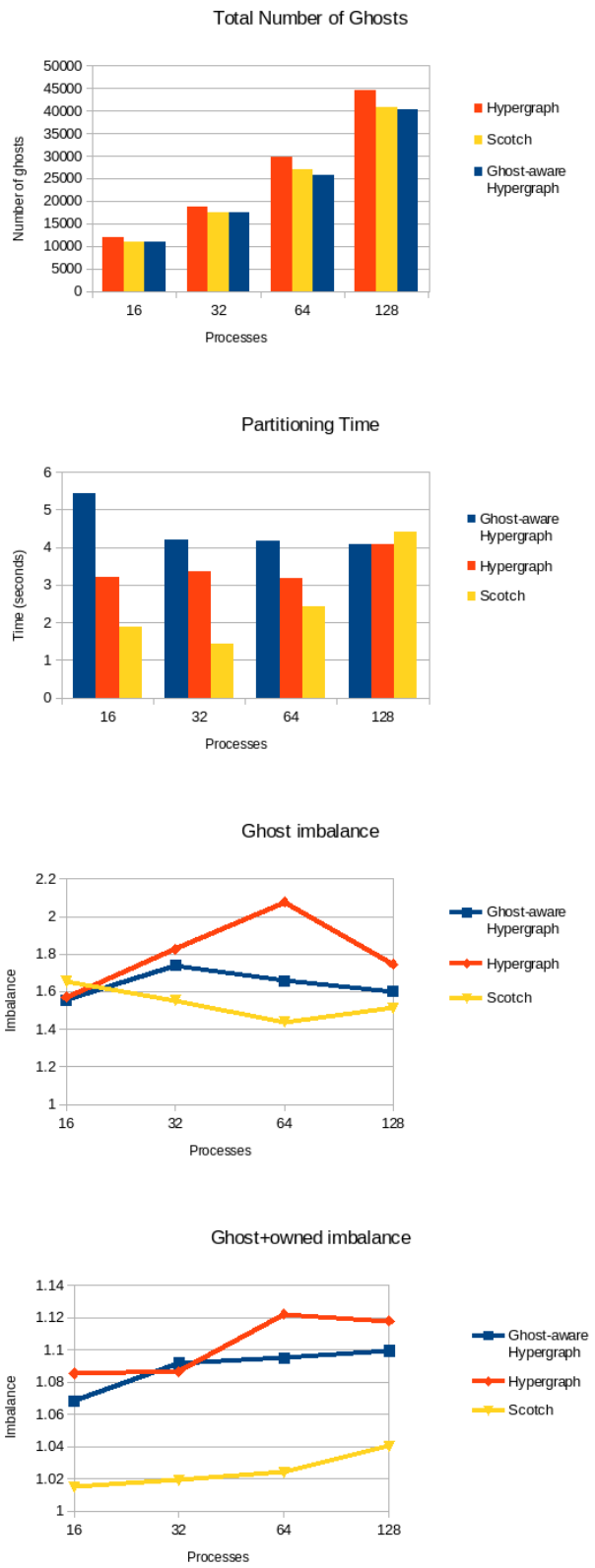


Fig. 6.3: Statistics from partitioning the North Atlantic Ocean mesh using 16, 32, 64, and 128 processes. (Top) The sum of the number of ghost elements per part. (Top Middle) The partitioning time. (Bottom Middle) The imbalance of ghosts elements from each partition. (Bottom) The imbalance of ghost+owned elements.

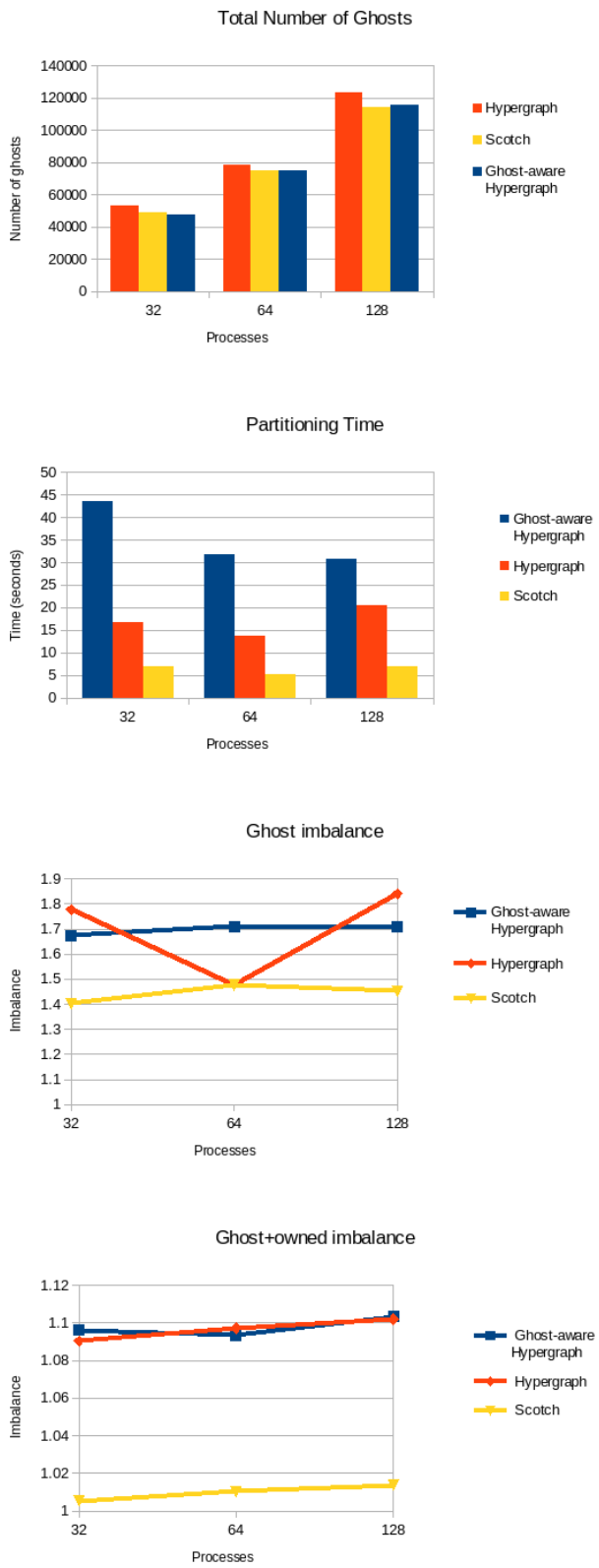


Fig. 6.4: Statistics from partitioning the 15km Ocean mesh using 32, 64, and 128 processes. (Top) The sum of the number of ghost elements per part. (Top Middle) The partitioning time. (Bottom Middle) The imbalance of ghosts elements from each partition. (Bottom) The imbalance of ghost+owned elements.

7. Future Work. With the new ParMA interface in Zoltan2, we would like to run some experiments with PULP [27] and ParMA to see how they compare. Also we would like to compare our hypergraph model of ghosting with the algorithm provided in [25]. This comparison would require the ocean depth information as weights to accurately compare the two algorithms.

We found that our ghost-aware results are promising and seem to agree with those in [25]. In their paper they found much better results in the ghost-aware model when the core counts scaled out to the thousands. Thus we need to see if our ghost-aware hypergraph model will follow the same pattern and see more significant improvements at larger core counts. Also further work should go into looking at the imbalance of the ghosts. We would like to test the ParMA ghost balancer to see if we can reduce the ghost imbalance without significantly increasing the number of ghosts.

8. Conclusions. In this work, we addressed two mesh-based applications whose performance depends on factors other than the element imbalance. We discussed partitioning algorithms that can target the different performance factors and demonstrated the use of graph and hypergraph partitioning on the two target problems. We also showed the use of ParMA as a local refinement tool to improve the resulting imbalance as a result of the partitioning.

For the PHASTA application we found that hypergraph partitioning can reduce the total number of vertices compared to graph partitioning by up to 4%. Also using ParMA, we can reduce the high vertex imbalance that is found in graph and hypergraph partitions with the trade-off of some increase in the total number of vertices in the mesh.

We developed a new ghost-aware hypergraph model to partition meshes for applications like the MPAS-Ocean which use two or more layers of ghost. We found this new model to have small improvements in the number of ghosts over the traditional graph approach but also tended to have a larger imbalance of the ghosts across the different processes. We predict that our ghost-aware model will perform better at larger scales, but require more testing to back up this hypothesis.

Acknowledgment. The authors thank Brent Perschbacher, Cameron Smith, Dan Ibanez, Andrew Bradley, and Andrew Salinger for helping the process of integrating SCOREC within Trilinos. Also we thank Erik Boman, Vitus Leung, Siva Rajamanickam, and Michael Wolf for their assistance in the Zoltan2 development.

REFERENCES

[1] *Zoltan2 Home Page*, 2015. `https://trilinos.org/packages/zoltan2/`.

[2] E. G. Boman, Ü. V. Çatalyürek, C. Chevalier, and K. D. Devine, *The Zoltan and Isorropia parallel toolkits for combinatorial scientific computing: Partitioning, ordering, and coloring*, Sci. Program., 20 (2012), pp. 129–150.

[3] T. Bui and C. Jones, *A heuristic for reducing fill in sparse matrix factorization*, in Proc. 6th SIAM Conf. Parallel Processing for Scientific Computing, SIAM, 1993, pp. 445–452.

[4] Ü. Çatalyürek and C. Aykanat, *Decomposing irregularly sparse matrices for parallel matrix-vector multiplications*, Lecture Notes in Computer Science, 1117 (1996), pp. 75 – 86.

[5] ———, *Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication*, IEEE Trans. Parallel Dist. Systems, 10 (1999), pp. 673–693.

[6] ———, *A fine-grain hypergraph model for 2D decomposition of sparse matrices*, in Proc. IPDPS 8th Int'l Workshop on Solving Irregularly Structured Problems in Parallel (Irregular 2001), April 2001.

[7] U. V. Catalyurek, E. G. Boman, K. D. Devine, D. Bozdağ, R. T. Heaphy, and L. A. Riesen, *A repartitioning hypergraph model for dynamic load balancing*, J. Parallel Distrib. Comput., 69 (2009), pp. 711–724.

[8] U. V. Çatalyürek, *Hypergraph Models for Sparse Matrix Partitioning and Reordering*, PhD thesis, Bilkent University, Computer Engineering and Information Science, Nov 1999.

[9] C. Chevalier and E. G. Boman, *An accurate hypergraph model for mesh partitioning*, Oct. 2009. Poster presented at the 2009 SIAM Workshop on Combinatorial Scientific Computing.

[10] K. C. Chitale, M. Rasquin, J. Martin, and K. Jensen, *Finite Element Flow Simulations of the EUROLIFT DLR-F11 High Lift Configuration*, AIAA Paper 2014-0749, (2014).

[11] K. C. Chitale, O. Sahni, M. S. Shephard, and K. Jansen, *Anisotropic Adaptation for Transonic Flows with Turbulent Boundary Layers*, AIAA, (2014).

[12] G. Cybenko, *Dynamic load balancing for distributed memory multiprocessors*, J. Parallel Distrib. Comput., 7 (1989), pp. 279–301.

[13] K. Devine, E. Boman, R. Heaphy, R. Bisseling, and U. Catalyurek, *Parallel hypergraph partitioning for scientific computing*, in Proc. of 20th International Parallel and Distributed Processing Symposium (IPDPS'06), IEEE, 2006.

[14] O. Fortmeier, H. Bücker, B. F. Auer, and R. Bisseling, *A new metric enabling an exact hypergraph model for the communication volume in distributed-memory parallel applications*, Parallel Computing, 39 (2013), pp. 319 – 335.

[15] B. Hendrickson and T. G. Kolda, *Graph partitioning models for parallel computing*, Parallel Computing, 26 (2000), pp. 1519 – 1534.

[16] B. Hendrickson and R. Leland, *A multilevel algorithm for partitioning graphs*, in Proc. Supercomputing '95, ACM, December 1995.

[17] M. A. Heroux, R. A. Bartlett, V. E. Howle, R. J. Hoekstra, J. J. Hu, T. G. Kolda, R. B. Lehoucq, K. R. Long, R. P. Pawlowski, E. T. Phipps, A. G. Salinger, H. K. Thornquist, R. S. Tuminaro, J. M. Willenbring, A. Williams, and K. S. Stanley, *An overview of the trilinos project*, ACM Trans. Math. Softw., 31 (2005), pp. 397–423.

[18] Y. F. Hu, R. J. Blake, and D. R. Emerson, *An optimal migration algorithm for dynamic load balancing*, Concurrency: Practice and Experience, 10 (1998), pp. 467 – 483.

[19] G. Karypis and V. Kumar, *A fast and high quality multilevel scheme for partitioning irregular graphs*, SIAM J. Scientific Computing, 20 (1998), pp. 359–392.

[20] G. Karypis and V. Kumar, *Multilevel K-way hypergraph partitioning*, VLSI Design, (2000).

[21] T. Lengauer, *Combinatorial Algorithms for Integrated Circuit Layout*, John Wiley & Sons, New York, NY, 1990.

[22] F. Pellegrini and J. Roman, *Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs*, in High-Performance Computing and Networking, Springer, 1996, pp. 493–498.

[23] M. Rasquin, C. Smith, K. Chitale, S. Seol, B. A. Matthews, J. L. Martin, O. Sahni, R. M. Loy, M. S. Shephard, and K. E. Jansen, *Scalable fully implicit finite element flow solver with application to high-fidelity flow control simulations on a realistic wing design*, Computing in Science and Engineering, (2014).

[24] T. Ringler, M. Petersen, R. L. Higdon, D. Jacobsen, P. W. Jones, and M. Maltrud, *Ocean Modelling*, Ocean Modelling, 69 (2013), pp. 211–232.

[25] A. Sarje, S. Song, D. Jacobsen, K. Huck, J. Hollingsworth, A. Malony, S. Williams, and L. Oliker, *Parallel performance optimizations on unstructured mesh-based simulations*, Procedia Computer Science, 51 (2015), pp. 2016 – 2025. International Conference On Computational Science, {ICCS} 2015Computational Science at the Gates of Nature.

[26] K. Schloegel, G. Karypis, and V. Kumar, *Multilevel diffusion algorithms for repartitioning of adaptive meshes*, J. Parallel Distrib. Comput., 47 (1997), pp. 109–124.

[27] G. M. Slota, K. Madduri, and S. Rajamanickam, *Pulp: Scalable multi-objective multi-constraint partitioning for small-world networks*, in Big Data (Big Data), 2014 IEEE International Conference on, IEEE, 2014, pp. 481–490.

[28] C. W. Smith, M. Rasquin, D. Ibanez, K. Jansen, and M. S. Shephard, *A Parallel Unstructured Mesh Infrastructure*, SIAM Journal on Scientific Computing, Submitted (2015).

[29] B. Vastenhouw and R. H. Bisseling, *A two-dimensional data distribution method for parallel sparse matrix-vector multiplication*, SIAM Review, 47 (2005), pp. 67–95.

[30] C. Walshaw, M. Cross, and M. G. Everett, *Parallel dynamic graph partitioning for adaptive unstructured meshes*, J. Parallel Distrib. Comput., 47 (1997), pp. 102–108.

[31] M. Willebeek-LeMair and A. Reeves, *Strategies for dynamic load balancing on highly parallel computers*, IEEE Parallel and Distrib. Sys., 4 (1993), pp. 979–993.

[32] M. Zhou, O. Sahni, M. S. Shephard, K. D. Devine, and K. Jansen, *Controlling unstructured mesh partitions for massively parallel simulations*, SIAM J. Sci. Comp., 32 (2010), pp. 3201–3227.

OPTIMIZATION OF BLOCK SPARSE MATRIX-VECTOR MULTIPLICATION ON SHARED-MEMORY PARALLEL ARCHITECTURES

RYAN A. EBERHARDT* AND MARK HOEMMEN†

Abstract. We examine the implementation of block CSR sparse matrix-vector multiplication (SpMV) for sparse matrices with dense block substructure (optimized for blocks with sizes from 2x2 to 32x32) on CPU, Intel many-integrated-core, and GPU architectures. Previous work has largely focused on the design of novel data structures to optimize performance for specific architectures or to store variable-sized, variably-aligned blocks; this paper instead seeks to optimize SpMV using the standard BCSR format, maintaining compatibility with existing preconditioners and solvers. We give a set of algorithms that offers an 80x speedup over Intel Math Kernel Library (MKL) and a 4x speedup over NVIDIA cuSPARSE.

1. Introduction. Sparse matrix-vector multiplication (SpMV) computes the operation $\boldsymbol{y} \leftarrow \alpha \boldsymbol{y} + \beta \boldsymbol{A}\boldsymbol{x}$, where $\boldsymbol{A}$ is a sparse matrix and $\boldsymbol{x}$ and $\boldsymbol{y}$ are dense vectors. It dominates the running time in many scientific and engineering applications and is notorious for sustaining low percentages of machine peak performance due to its memory-bound nature. Improving performance of SpMV generally requires selecting appropriate data structure transformations and memory access patterns for the matrices being used and the underlying hardware architecture. Sparse matrices arising from finite-element analysis often exhibit a dense block substructure, as do matrices from discretizations in which each node in a graph has multiple degrees of freedom. This block substructure can be exploited to represent a sparse matrix with less space. Block Compressed Sparse Row (BCSR) is the most popular format for representing general sparse matrices with constant-sized blocks; it achieves high performance in the general case without being overly dependent on matrix structure.

This paper contributes algorithms for the efficient execution of SpMV using BCSR on shared-memory parallel architectures, including traditional CPU architectures (e.g., Sandy Bridge), the Intel Knights Corner (KNC) many-integrated-core architecture, and NVIDIA GPU architectures.

2. Related Work. A great amount of literature exists regarding the optimization of SpMV for various parallel architectures using standard non-block formats. Williams et al. investigate the tuning of CSR SpMV on AMD and Intel CPU architectures (among others) [8], and Bell and Garland compare the performance of SpMV using varying sparse matrix formats on NVIDIA GPUs [1]. Some have explored algorithms that reorder matrices in order to *create* a dense block substructure [7][3], and others have created novel data structures optimized for unaligned blocks or specific hardware architectures [4][5]. However, little research has been published regarding the optimization of SpMV algorithms for the Block CSR format on highly parallel architectures. We seek to create general-purpose algorithms compatible with existing solvers and preconditioners for the BCSR format.

3. BCSR SpMV on Shared-Memory Parallel Architectures.

3.1. Storage Format – Block CSR. The algorithms presented in this paper use sparse matrices stored in the Block Compressed Sparse Row (BCSR/BSR) format with column-major blocks. BCSR is the most popular blocked sparse matrix storage format, as it performs well on nearly any matrix, is not highly dependent on matrix structure, and is a simple data structure to construct.

*William Rainey Harper College, r_eberhardt4@mail.harpercollege.edu
†Sandia National Laboratories, mhoemme@sandia.gov

BCSR stores nonzero blocks contiguously, row by row, in an array *val* of length $nnzb \cdot bs^2$; for each nonzero block, an entry is added to an array *col_idx* of length *nnzb* with its column index. An array *row_ptr* stores $m + 1$ values, where each value stores the index of a row's first block in *val* and *col_idx*. For a more in-depth description of the Block CSR format, please refer to Richard Vuduc's Ph.D. thesis [6].

We have elected to use column-major blocks in order to improve temporal cache locality for $\boldsymbol{x}$. By streaming through columns of *val*, we access $\boldsymbol{x}$ consecutively; a row-major layout would access segments of *val* multiple times, causing L1 evictions for larger blocks. Additionally, using column-major blocks simplifies the reductions for the GPU algorithms presented in Sections 3.4.1 and 3.4.2.

3.2. Primary Algorithm. In our algorithm, we assign a parallel worker (a thread or core for CPU and CPU-like architectures, or a warp of threads for GPU architectures) to one or more block rows. In each of its block rows, a parallel worker iterates down each column, across the block row, in order to achieve streaming access to the *val* array (which has column-major blocks stored row-wise). The manner in which a parallel worker does this is tuned for different hardware architectures (e.g., for GPU architectures, each warp has threads enabling it to iterate through multiple columns at a time) and is described in Sections 3.3 and 3.4.

3.3. CPU and Many-Integrated-Core Architectures. For these architectures, we delegate the block rows of the sparse matrix across n threads, such that each thread is responsible for approximately mb/n block rows.

Each thread iterates over its block rows serially; for each block row, a thread retrieves pointers to the start and end of the row in the *val* and *col_idx* arrays by retrieving $row\_ptr[thread\_idx]$ and $row\_ptr[thread\_idx{+}1]$. It then iterates from this first block pointer to the last block pointer. In each block iteration, the algorithm finds the column index of the block in $col\_idx[block\_ptr]$ and iterates over the columns within the block. In each column iteration, the algorithm retrieves $\boldsymbol{x}_{block,col}$, then iterates through each element in the column, multiplying by $\boldsymbol{x}_{block,col}$ and updating a temporary output array in registers. After processing the block row, the thread updates the output array $\boldsymbol{y}$. No interactions between threads are present.

A pseudocode implementation of this algorithm is shown in Algorithm 5 and a diagram of the memory access pattern is shown in Figure 3.1.

Algorithm 5 BCSR SpMV kernel for CPU architectures. r and c denote a thread's position (row, column) within a block.

```
target_block_row ← thread_idx
first_block ← row_ptr[target_block]
last_block ← row_ptr[target_block + 1]
local_out[bs] ← {0}
for block ← first_block; block < last_block; block++ do
    jb ← col_idx[block]
    for c ← 0; c < bs; c++ do
        vec_this_col ← vec[jb][c]
        for r ← 0; r < bs; r++ do
            local_out[r] += val[block][c][r] * vec_this_col
for r ← 0; r < bs; r++ do
    global_out[target_block_row][r] += local_out[r]
```

$$\begin{bmatrix} \begin{bmatrix} \text{T0, I0} & \text{T0, I1} \\ \text{T0, I2} & \text{T0, I3} \end{bmatrix} \begin{bmatrix} \text{T0, I4} & \text{T0, I5} \\ \text{T0, I6} & \text{T0, I7} \end{bmatrix} \begin{bmatrix} \text{T0, I8} & \text{T0, I9} \\ \text{T0, I10} & \text{T0, I11} \end{bmatrix} \\ \begin{bmatrix} \text{T1, I0} & \text{T1, I1} \\ \text{T1, I2} & \text{T1, I3} \end{bmatrix} \begin{bmatrix} \text{T1, I4} & \text{T1, I5} \\ \text{T1, I6} & \text{T1, I7} \end{bmatrix} \begin{bmatrix} \text{T1, I8} & \text{T1, I9} \\ \text{T1, I10} & \text{T1, I11} \end{bmatrix} \\ \begin{bmatrix} \text{T2, I0} & \text{T2, I1} \\ \text{T2, I2} & \text{T2, I3} \end{bmatrix} \begin{bmatrix} \text{T2, I4} & \text{T2, I5} \\ \text{T2, I6} & \text{T2, I7} \end{bmatrix} \begin{bmatrix} \text{T2, I8} & \text{T2, I9} \\ \text{T2, I10} & \text{T2, I11} \end{bmatrix} \end{bmatrix}$$

Fig. 3.1: Matrix access pattern for CPU architectures. T_a, I_b indicates thread a, iteration b. Each thread is responsible for one block row.

Memory access patterns are highly optimal for this algorithm. Each thread achieves streaming access to *val*; the access pattern is also predictable by a hardware prefetcher, so access latencies are reduced. If the architecture offers a large cache and *bs* is small, the algorithm also achieves streaming access to $\boldsymbol{x}$ and *col_idx*. *row_ptr* is accessed only twice per thread, so its memory access cost is generally insignificant.

This algorithm is also friendly to SIMD vectorization. If *bs* is known at compile time, the innermost loop over the elements in a column has compile-time bounds and can be vectorized by the compiler. In our tests, vectorization provided up to 4.7x the performance of non-vectorized code on KNC.

Reuse of $\boldsymbol{x}$ is limited; though better than naive non-block CSR, $\boldsymbol{x}$ is reused a maximum of *bs* times. Performance could be improved by tiling blocks to improve temporal locality; this possibility is discussed in Section 5, Conclusions and Future Work.

3.4. GPU Architectures. In this section, we seek to adapt the previous algorithm for highly multi-threaded GPU architectures. The CUDA programming model operates with a large number of threads operating in SIMT (single instruction, multiple thread) style, where a group of 32 threads (known as a warp) concurrently execute the same instruction. The threads in a warp must work cooperatively; if threads diverge and issue different sets of instructions, the warp scheduler will mask off parts of the warp, executing different branch paths separately and reducing the rate at which instructions can be issued.

For optimal performance, the threads in a warp must also access contiguous segments of memory. NVIDIA devices have a global memory bus width of 128 bytes, which can hold 16 8-byte double-precision values. If 16 threads access a sequential range of doubles that lie in a 128-byte row of DRAM in the same SIMD instruction issue, the memory accesses is "coalesced" into a single memory access instruction.

In the following three sections, we propose algorithms that each assign a warp to a single block row, but assign the threads within a warp to the elements in a block row in different ways based on the block size of the matrix.

3.4.1. Block row per warp, operating by block. In this algorithm, a warp is assigned to a single block row, and threads in the warp are assigned to elements in the block row column-wise (see Figure 3.2 for an illustration of thread assignment). To cover the entire block row, a warp iterates over the row's blocks, handling $bs^2/32$ blocks at a time, where *bs* is the block size. This is the number of blocks that a warp can cover completely if each thread is assigned to a single element. For example, with a block size of 3, the warp can cover 3 complete blocks (27 elements) at a time. The warp will only cover complete blocks; in the case of 3x3 blocks, five threads will be inactive in each iteration $(32 - (3 \cdot 3) \cdot 3)$.

Because a warp only covers complete blocks, a thread's assigned position within a block will never change between iterations. On initialization, a thread calculates the index of the block row it is targeting, finds pointers to the start and end of its row from *row_ptr*, and finds a pointer to the block it should begin its work on using the row start pointer and its lane number. It also calculates its assigned position within a block (r and c). It then iterates through the block row, beginning at its assigned block and advancing by $bs^2/32$ blocks at a time (the number of complete blocks that the warp can cover), multiplying its target element in $\boldsymbol{A}$ by the corresponding value from $\boldsymbol{x}$ and adding the product to a register that it uses to maintain a running total. Once the threads in a warp have iterated through a block row, threads that covered the same vertical (r) position in a block reduce the values stored in their local registers and write the reduced output to global memory (Figure 3.3).

This algorithm is described in pseudocode in Algorithm 6, and its memory access pattern is illustrated in Figure 3.2.

Algorithm 6 Block-by-block BCSR SpMV kernel for GPU architectures. r and c denote a thread's position (row, column) within a block.

```
bs ← block size
target_block_row ← (thread_block_idx * thread_block_dim + thread_idx)/32
lane ← thread_idx%32
first_block ← row_ptr[target_block]
last_block ← row_ptr[target_block + 1]
target_block ← first_block + lane/(bs · bs)
c ← (lane/bs)%bs
r ← lane%bs
▷ Shared memory for reduction step:
shared_out ← <alloc thread_block_size · sizeof(double) bytes shared mem>
shared_out[thread_idx] ← 0
if lane < 32/(bs * bs) * (bs * bs) then                ▷ Only process whole blocks
    ▷ Iterate through block row:
    local_out ← 0
    for ; target_block < last_block; target_block += 32/(bs · bs) do
        x_elem ← x[col_ind[target_block]][c]
        A_elem ← A[target_block][c][r]
        local_out += x_elem · A_elem
    ▷ Reduction:
    shared_out ← local_out
    stride ← round_up_to_power_of_two((32/bs)/2)
    for ; stride ≥ 1; stride /= 2 do
        if lane < stride · bs && lane + stride · bs < 32 then
            shared_out[thread_idx] += shared_out[thread_idx + stride · bs]
    ▷ Write reduced value to global memory:
    if lane < bs then
        global_out[target_block_row][lane] += shared_out[thread_idx]
```

This algorithm exhibits high-performing memory access patterns for *val* and $\boldsymbol{x}$. Accesses to *val* will be fully coalesced. Accesses to $\boldsymbol{x}$ are fully coalesced when the block size is 2 and partially coalesced for larger block sizes. Accesses to *row_ptr* and writes to *global_out* are generally not coalesced, but these transactions represent an insignificant portion of all memory operations.

$$
\begin{bmatrix}
\begin{bmatrix} \text{T0} & \text{T2} \\ \text{T1} & \text{T3} \end{bmatrix}
\begin{bmatrix} \text{T4} & \text{T6} \\ \text{T5} & \text{T7} \end{bmatrix}
\begin{bmatrix} \text{T0} & \text{T2} \\ \text{T1} & \text{T3} \end{bmatrix} & & & \\
\begin{bmatrix} \text{T8} & \text{T10} \\ \text{T9} & \text{T11} \end{bmatrix}
\begin{bmatrix} \text{T12} & \text{T14} \\ \text{T13} & \text{T15} \end{bmatrix}
\begin{bmatrix} \text{T8} & \text{T10} \\ \text{T9} & \text{T11} \end{bmatrix}
\begin{bmatrix} \text{T12} & \text{T14} \\ \text{T13} & \text{T15} \end{bmatrix}
\begin{bmatrix} \text{T8} & \text{T10} \\ \text{T9} & \text{T11} \end{bmatrix} & & \\
\begin{bmatrix} \text{T16} & \text{T18} \\ \text{T17} & \text{T19} \end{bmatrix}
\begin{bmatrix} \text{T20} & \text{T22} \\ \text{T21} & \text{T23} \end{bmatrix}
\begin{bmatrix} \text{T16} & \text{T18} \\ \text{T17} & \text{T19} \end{bmatrix}
\begin{bmatrix} \text{T20} & \text{T22} \\ \text{T21} & \text{T23} \end{bmatrix} & & &
\end{bmatrix}
$$

1st iteration 2nd iteration 3rd iteration

Fig. 3.2: Matrix access pattern of the by-block algorithm for a matrix with 2x2 blocks. The warp size has been reduced to 8 for the purposes of visualization (though in practice, the warp size will always be 8). Warp 0 (threads 0-7) handles block row 0, warp 1 (threads 8-15) handles block row 1, and warp 2 (threads 16-23) handles block row 2.

$$
\begin{bmatrix}
\begin{bmatrix} y_{01} \\ y_{02} \end{bmatrix} \\
\begin{bmatrix} y_{11} \\ y_{12} \end{bmatrix} \\
\begin{bmatrix} y_{21} \\ y_{22} \end{bmatrix}
\end{bmatrix}
\begin{array}{l}
\longleftarrow \text{T0, T2, T4, T6} \\
\longleftarrow \text{T1, T3, T5, T7} \\
\longleftarrow \text{T8, T10, T12, T14} \\
\longleftarrow \text{T9, T11, T13, T15} \\
\longleftarrow \text{T16, T18, T20, T22} \\
\longleftarrow \text{T17, T19, T21, T23}
\end{array}
$$

Fig. 3.3: Warp reduction for the matrix-vector multiplication example shown in Figure 3.2. Each thread in a warp reduces with other threads that also covered the same row within a block; the output is then written to global memory.

$$
\begin{bmatrix}
\begin{bmatrix} \text{T0} & \text{T3} & \text{T6} \\ \text{T1} & \text{T4} & \text{T7} \\ \text{T2} & \text{T5} & \text{T8} \end{bmatrix}
\begin{bmatrix} \text{T9} & \text{T12} & \text{T15} \\ \text{T10} & \text{T13} & \text{T16} \\ \text{T11} & \text{T14} & \text{T17} \end{bmatrix}
\begin{bmatrix} \text{T18} & \text{T21} & \text{T24} \\ \text{T19} & \text{T22} & \text{T25} \\ \text{T20} & \text{T23} & \text{T26} \end{bmatrix}
\begin{bmatrix} \text{T0} & \text{T3} & \text{T6} \\ \text{T1} & \text{T4} & \text{T7} \\ \text{T2} & \text{T5} & \text{T8} \end{bmatrix} \\
\begin{bmatrix} \text{T32} & \text{T35} & \text{T38} \\ \text{T33} & \text{T36} & \text{T39} \\ \text{T34} & \text{T37} & \text{T40} \end{bmatrix}
\begin{bmatrix} \text{T41} & \text{T44} & \text{T47} \\ \text{T42} & \text{T45} & \text{T48} \\ \text{T43} & \text{T46} & \text{T49} \end{bmatrix}
\begin{bmatrix} \text{T50} & \text{T53} & \text{T56} \\ \text{T51} & \text{T54} & \text{T57} \\ \text{T52} & \text{T55} & \text{T58} \end{bmatrix}
\begin{bmatrix} \text{T32} & \text{T35} & \text{T38} \\ \text{T33} & \text{T36} & \text{T39} \\ \text{T34} & \text{T37} & \text{T40} \end{bmatrix} \\
\begin{bmatrix} \text{T64} & \text{T67} & \text{T70} \\ \text{T65} & \text{T68} & \text{T71} \\ \text{T66} & \text{T69} & \text{T72} \end{bmatrix}
\begin{bmatrix} \text{T73} & \text{T76} & \text{T79} \\ \text{T74} & \text{T77} & \text{T80} \\ \text{T75} & \text{T78} & \text{T81} \end{bmatrix}
\begin{bmatrix} \text{T82} & \text{T85} & \text{T88} \\ \text{T83} & \text{T86} & \text{T89} \\ \text{T84} & \text{T87} & \text{T90} \end{bmatrix}
\end{bmatrix}
$$

1st iteration 2nd iteration

Fig. 3.4: Matrix access pattern of the by-block algorithm for a matrix with 3x3 blocks. Note that five threads per warp (e.g., warp 0, threads 27-31) are left inactive, as they do not cover a complete block.

Unfortunately, accesses to *col_idx* are generally not coalesced, and since the values in *col_idx* are required to access $\boldsymbol{x}$, this adds latency to those memory transactions. However, as the algorithm uses few registers, many warps may fit into the streaming multiprocessors, so the GPU can generally hide the latency and keep the memory bus saturated by switching between warps.

Each thread accumulates a result in a register and there is no interaction between threads until the final reduction, so latency is low. The reduction always occurs within a warp, so no synchronization primitives are required.

While this algorithm theoretically achieves 100% theoretical thread utilization for block sizes that are powers of two (because warps can cover blocks evenly, with no threads leftover), it potentially produces a large number of inactive threads for other block sizes. Consider the case of 3x3 blocks; a warp can cover only 3 full blocks (27 elements), leaving 5 threads in each warp (15.6%) inactive. (See Figure 3.4.) The algorithm uses few registers and achieves high occupancy, so it is generally able to issue enough instructions from a large number of warps to keep the memory bus saturated despite this decrease in active threads. However, this is an important consideration, and performance does decrease relative to the algorithm described in the next section (which is able to handle these cases more efficiently) for 3x3 and 5x5 blocks (see Figure 4.4).

A more significant problem is the block size limitation that the whole-block constraint imposes. Because a warp of only 32 threads must cover an entire block, this algorithm cannot multiply matrices that have block sizes larger than 5. This motivates the design of a similar algorithm that better handles large block sizes.

3.4.2. Block row per warp, operating by column. This algorithm is similar to the previous one, but relaxes the requirement that warps cover whole blocks. Instead, warps cover whole columns, enabling larger block sizes (up to 32x32, where a warp would cover a single column) and a more efficient handling of matrices with block sizes that are not powers of two. Unlike the previous algorithm, a warp iterates through its block row's *columns*, covering $bs/32$ columns at a time. With the whole-block requirement replaced with a whole-column requirement, the assigned vertical position of a thread within a block will never change, but a thread must compute its target block and target column in every iteration. The algorithm is shown in pseudocode in Algorithm 7 and its memory access pattern for 3x3 blocks – improved over that of the previous algorithm – is shown in Figure 3.5.

Like the previous block-by-block algorithm, this algorithm has minimal interaction between threads and achieves coalesced or partially coalesced accesses to *val* and $\boldsymbol{x}$, but it is now able to handle large block sizes more efficiently. For the case of a matrix with 3x3 blocks, this algorithm has only 2 inactive threads (an improvement from the 5 inactive threads of the previous algorithm for this scenario). However, this comes at the cost of increased latency from integer operations. The algorithm must compute a block index and a horizontal position within that block, and the memory requests stall on these operations. By maximizing occupancy, we are usually able to minimize this additional latency, but the previous algorithm tends to outperform this one for block sizes up to 5x5.

3.4.3. Row-per-thread. When block sizes are sufficiently large, a simpler, more efficient algorithm may be introduced. For these cases, a CUDA thread block is assigned to a block row, and each thread within a thread block is responsible for a single row within the block row. A thread iterates through its assigned (non-block) row, multiplying each element by the appropriate value from $\boldsymbol{x}$ and accumulating the results in a register. A simple version of this algorithm is shown in Algorithm 8.

This algorithm achieves fully-coalesced accesses to *val* when the block size is a multiple

Algorithm 7 Column-by-column BCSR SpMV kernel for GPU architectures.

```
1:  bs ← block size
2:  target_block_row ← (thread_block_idx * thread_block_dim + thread_idx) / 32
3:  lane ← thread_idx % 32
4:  first_block ← row_ptr[target_block]
5:  last_block ← row_ptr[target_block + 1]
6:  target_col ← first_block · bs + lane / bs
7:  r ← lane % bs
    ▷ Shared memory for reduction step:
8:  shared_out ← <alloc thread_block_size · sizeof(double) bytes shared mem>
9:  shared_out[thread_idx] ← 0
10: if lane < (32 / bs) * bs then                    ▷ Only process whole columns
        ▷ Iterate through columns:
11:     local_out ← 0
12:     for ; target_col < last_block · bs; target_col += 32 / bs do
13:         block ← target_col / bs
14:         c ← target_col % bs
15:         A_elem ← A[block][c][r]
16:         x_elem ← x[col_ind[block]][c]
17:         local_out += x_elem · A_elem
        ▷ Reduction:
18:     shared_out ← local_out
19:     stride ← round_up_to_power_of_two((32 / bs) / 2)
20:     for ; stride ≥ 1; stride /= 2 do
21:         if lane < stride · bs && lane + stride · bs < 32 then
22:             shared_out[thread_idx] += shared_out[thread_idx + stride · bs]
        ▷ Write reduced value to global memory:
23:     if lane < bs then
24:         global_out[target_block_row][lane] += shared_out[thread_idx]
```

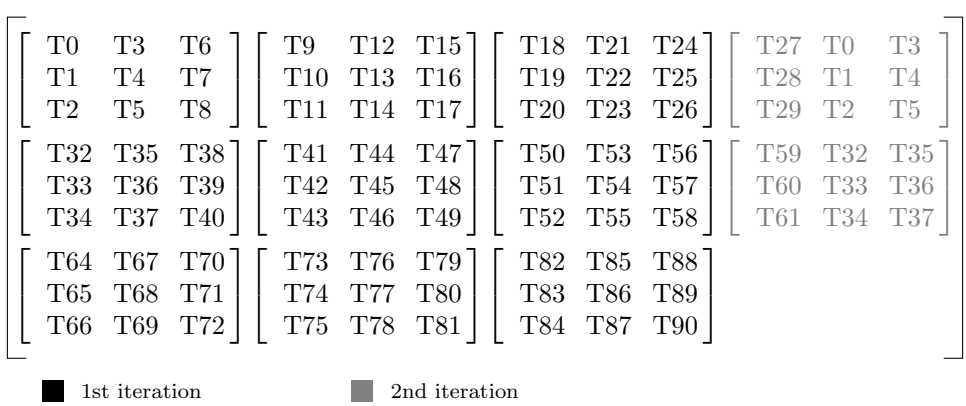


Fig. 3.5: Matrix access pattern of the by-column algorithm for a matrix with 3x3 blocks. Note that for this case, only two two threads per warp (e.g., warp 0, threads 30-31) are left inactive, an improvement over the by-block algorithm.

Algorithm 8 Simple row-per-thread BCSR SpMV kernel.

```
1: bs ← block size
2: target_block_row ← thread_block_idx
3: r ← lane ← thread_idx
4: first_block ← row_ptr[target_block_row]
5: last_block ← row_ptr[target_block_row + 1]
6: if r < bs then
7:     local_out ← 0
8:     for block ← first_block; block < last_block; block++ do
9:         for c ← 0; c < bs; c++ do
10:            local_out += x[col_ind[block]][c] · A.val[block][c][r]
11:    global_out[target_block_row][r] ← local_out
```

of 16 and partially-coalesced accesses when this is not the case. As threads use few registers, depend on little arithmetic for memory requests, and do not interact with other threads, occupancy is high and latency is low. Additionally, the inner loop can be unrolled for decreased instruction traffic and increased instruction-level parallelism.

In the basic implementation shown in Algorithm 8, access to $\boldsymbol{x}$ is not coalesced. This problem can be remedied by loading a portion of $\boldsymbol{x}$ into shared memory in a fully-coalesced fashion, where it can be reused by all threads in a thread block. An implementation of this is shown in Algorithm 9. Using shared memory in this way requires two barrier synchronizations for every block iteration, but we found this cost not to be significant.

Algorithm 9 Improved row-per-thread BCSR SpMV kernel with shared memory for $\boldsymbol{x}$.

```
1: bs ← block size
2: target_block_row ← thread_block_idx
3: r ← lane ← thread_idx
4: first_block ← row_ptr[target_block_row]
5: last_block ← row_ptr[target_block_row + 1]
6: shared_x ← <alloc thread_block_size · sizeof(double) bytes shared mem>
7: if r < bs then
8:     local_out ← 0
9:     for block ← first_block; block < last_block; block++ do
10:        Barrier synchronization
11:        shared_x[thread_idx] ← x[col_ind[block]][thread_idx]
12:        Barrier synchronization
13:        for c ← 0; c < bs; c++ do
14:            local_out += shared_x[c] · A.val[block][c][r]
15:    global_out[target_block_row][r] ← local_out
```

We find that this algorithm performs best for block sizes ≥ 16, as access to *val* and $\boldsymbol{x}$ have the opportunity to be fully-coalesced. See Figure 4.4.

The performance of this algorithm would seem to depend heavily on how well the matrix block size is matched to the thread block size; since the thread block size must be a multiple of 32, it would seem that the algorithm would perform poorly for a block size of 33, where 31 of 64 threads would be inactive. While performance certainly drops in this case, we did not notice as great of a performance impact as expected. The algorithm appears to have sufficient occupancy and instruction-level parallelism to hide latency despite the drop in

active threads.

4. Experimental Results. In this section, we examine the performance of our algorithms and compare them against vendor implementations (Intel Math Kernel Library and NVIDIA cuSPARSE) on Intel Sandy Bridge, Intel Knights Corner, and NVIDIA Kepler architectures.

Experiments with Intel hardware were run on the Sandia National Laboratories Compton test bed with two 8-core Sandy Bridge Xeon E5-2670 processors running at 2.6GHz and a KNC Xeon Phi 3120A card. Intel ICC 16.0.0, MKL 11.3 pre-release, and hwloc 1.6.2 were used. Experiments with NVIDIA hardware were run on the Shannon test bed with an NVIDIA K80S dual-GPU card (only one GPU on the card was used). GCC 4.9.0 and CUDA 7.0.18 were used. Implementations of the GPU algorithms used a texture cache to optimize accesses to $\boldsymbol{x}$. Test matrices were obtained from the University of Florida Sparse Matrix Collection [2] and are shown in Table 4.1. Matrices were selected from a variety of real-world applications.

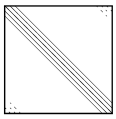
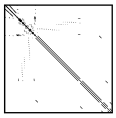
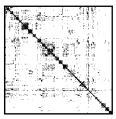
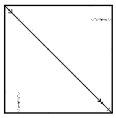
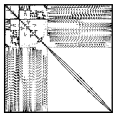
Plot	Name	bs	Dimensions (in blocks)	nnzb (nnzb/row)	Description
	GT01R	5	1.60K x 1.60K	20.37K (13)	2D inviscid fluid
	raefsky4	3	6.59K x 6.59K	111.4K (17)	Container buckling problem
	bmw7st_1	6	23.6K x 23.6K	229.1K (10)	Car body analysis
	pwtk	6	36.3K x 36.3K	289.3K (8)	Pressurized wind tunnel
	RM07R	7	545K x 545K	1.504M (28)	3D viscous turbulence
	audikw_1	3	314K x 314K	4.471M (14)	AUDI crankshaft model

Table 4.1: Overview of test block matrices used in experimental evaluation.

To understand how the algorithms perform on different architectures with matrices of different block sizes, we partitioned the raefsky4 test matrix (a finite-element static analysis of the buckling of a container) into blocks with sizes from 2 to 32, placing nonzero elements in the appropriate aligned blocks and filling in zeros. The results are shown in Figure 4.4. Performance of our algorithm on KNC peaks when the block size is 8 or 16, as the compiler perfectly vectorizes the code. On Kepler, we find that our by-block algorithm performs best

for the small block sizes it can handle, our by-column algorithm performs best for block sizes up to 16, and after that point, our row-per-thread algorithm – designed to handle large block sizes – performs best.

We also examined how our algorithm scales over a varying number of cores within a single node. For Sandy Bridge, we tested the algorithm with up to 8 cores on a single socket, and used two sockets to test with up to 16 cores. Results are shown in Figure 4.5. Performance appears to scale nearly linearly with additional cores within a node, even across sockets. While we performed no tests with multiple nodes, we expect this algorithm will continue to scale well due to a lack of inter-thread communication.

We found that enabling hyperthreading on Sandy Bridge did not yield significant benefits and actually reduced performance in some cases. However, using two hardware threads with the KNC architecture improved performance, likely due to a KNC in-order core's inability to issue instructions quickly enough. Using four hardware threads provided no significant increase in performance over two hardware threads.

We partitioned our test matrices into their dominant block sizes (see Table 4.1) and compared our algorithms to vendor implementations. We measured the mean time kernels took to execute and divided by the amount of unique data read (total size of $\boldsymbol{x}$, val, col_idx, and row_ptr) to determine the achieved throughput. Results are shown in Figures 4.1, 4.2, and 4.3. In general, we find that the algorithms performed best with larger blocks (as they were able to achieve better reuse of $\boldsymbol{x}$) and with larger matrices, as the time spent iterating through rows represented more of the total time than the fixed setup cost for each thread.

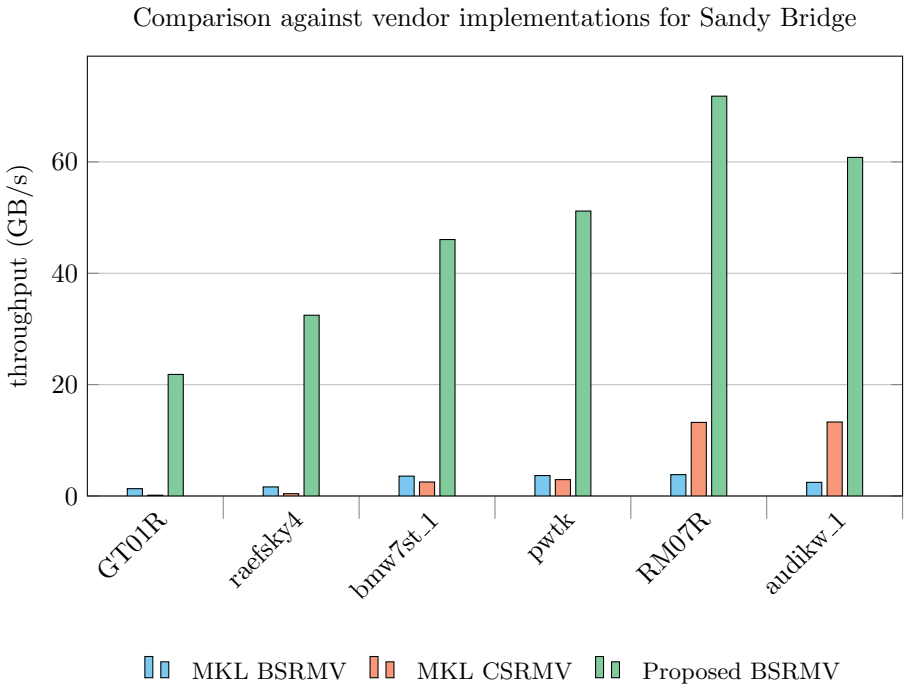


Fig. 4.1: Performance comparison on the Sandy Bridge CPU architecture for various test matrices using each matrix's dominant block size (see Table 4.1).

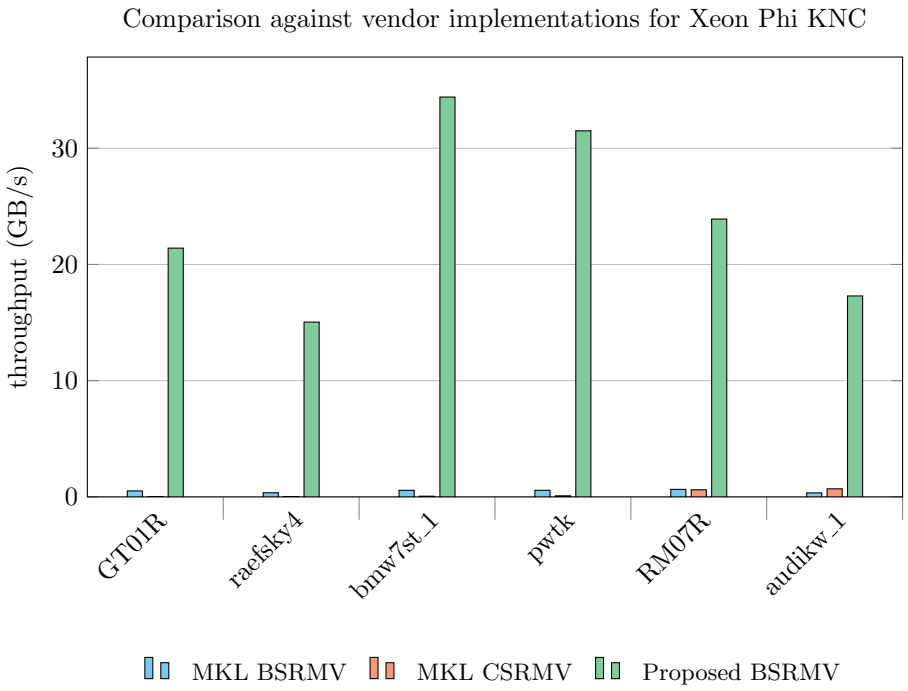


Fig. 4.2: Performance comparison on the Knights Corner (KNC) Xeon Phi architecture for various test matrices using each matrix's dominant block size (see Table 4.1).

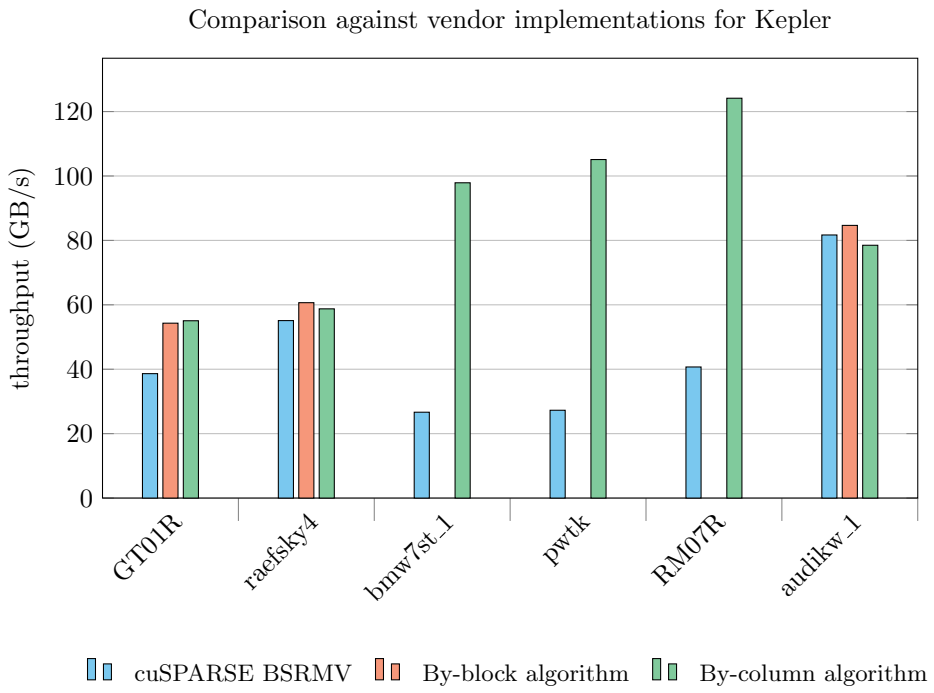


Fig. 4.3: Performance comparison on the Kepler GPU architecture for various test matrices using each matrix's dominant block size (see Table 4.1). The blocks of bmw7st_1, pwtk, and RM07R are larger than 5x5, so they cannot be handled by the by-block algorithm. No matrices in our test set had blocks large enough to justify the use of the row-per-thread algorithm, so we have omitted it from this comparison.

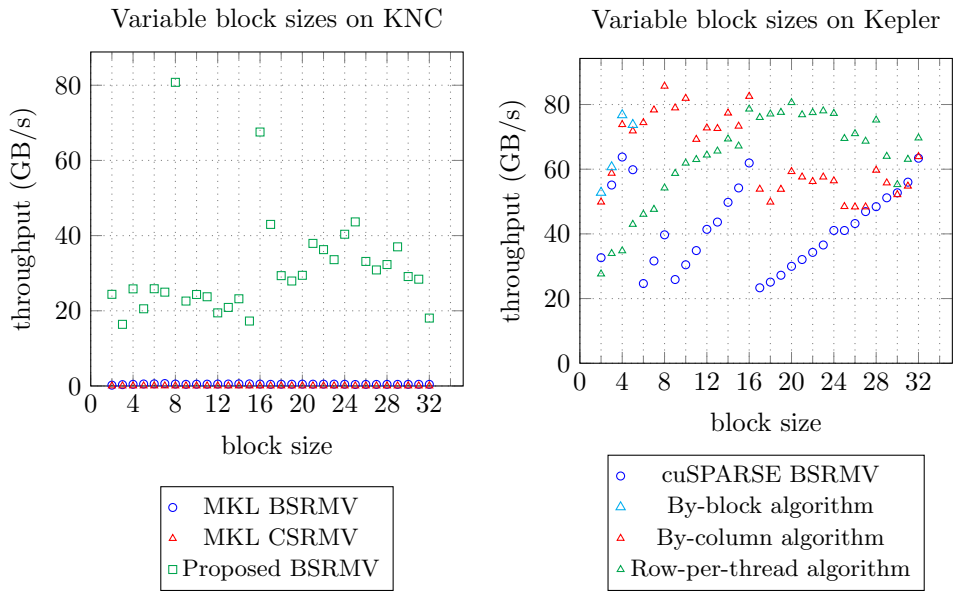


Fig. 4.4: Performance of SpMV algorithms running on the Xeon Phi KNC and Kepler GPU architectures for the raefsky4 test matrix, divided into variable block sizes.

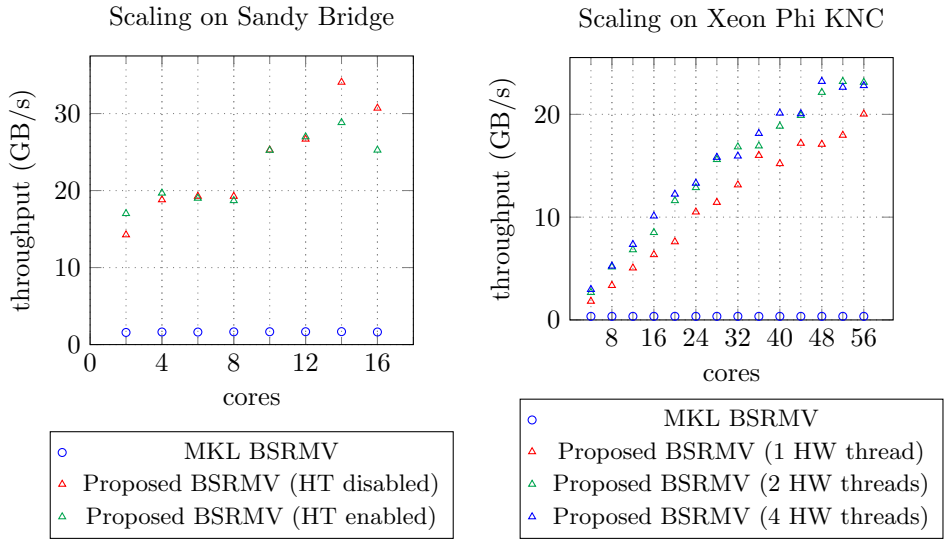


Fig. 4.5: Scaling of our algorithm on a Sandy Bridge Xeon E5-2670 processor and a KNC Xeon Phi using the raefsky4 test matrix. We tested the impact of hyperthreading (HT) on the CPU and the impact of hardware (HW) thread usage on the Xeon Phi.

5. Conclusions and Future Work. By optimizing memory access patterns and minimizing visible latencies, the algorithms presented in this paper are able to outperform the vendor-provided block sparse matrix-vector implementations and achieve high bandwidth utilization. However, especially on KNC, additional optimizations may be made to further improve the throughput of the algorithms; in many cases, the presented algorithm currently achieves only 10-20% of the advertised Xeon Phi bandwidth. In particular, a cooperative threading strategy may be developed so that KNC hardware threads may work on consecutive block rows and achieve increased temporal cache locality with $\boldsymbol{x}$. As KNC is an in-order architecture and this severely limits instruction throughput, a better thread cooperation strategy will benefit performance.

To improve performance for iterative solvers when nonzero blocks are unevenly distributed between rows, a preliminary analysis stage may be introduced in which the algorithm groups rows by $nnzb/row$. The multiplication can then be executed in a multi-pass style, where each pass includes rows of a certain length, in order to better distribute load between cores.

Data structure transformations may be required to further improve performance by a significant margin. Specifically, blocks may be grouped into and processed as tiles in order to potentially (a) further reduce the sizes of the *row_ptr* and *col_idx* arrays and (b) improve cache performance for $\boldsymbol{x}$. Such an optimization may be possible without changing the basic BCRS format by adding metadata pointing to tiles in the matrix.

6. Acknowledgements. We would like to thank Travis Fisher for reviewing a draft of this paper. We would also like to thank Carter Edwards, Simon Hammond, and Christian Trott for reviewing our work and offering technical assistance.

REFERENCES

[1] N. Bell and M. Garland, *Implementing sparse matrix-vector multiplication on throughput-oriented processors*, in Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, ACM, 2009, p. 18.

[2] T. Davis and Y. Hu, *The University of Florida Sparse Matrix Collection*, ACM Transactions on Mathematical Software, 38 (2011), pp. 1:1 – 1:25.

[3] V. Eijkhout, *Automatic determination of matrix blocks*, tech. rep., Department of Computer Science, University of Tennessee, 2001. LAPACK Working Note 151, 2001.

[4] X. Liu, M. Smelyanskiy, E. Chow, and P. Dubey, *Efficient sparse matrix-vector multiplication on x86-based many-core processors*, in Proceedings of the 27th international ACM conference on International conference on supercomputing, ACM, 2013, pp. 273–282.

[5] R. Shahnaz and A. Usman, *Blocked-based sparse matrix-vector multiplication on distributed memory parallel computers.*, Int. Arab J. Inf. Technol., 8 (2011), pp. 130–136.

[6] R. W. Vuduc, *Automatic performance tuning of sparse matrix kernels*, PhD thesis, Citeseer, 2003.

[7] R. W. Vuduc and H.-J. Moon, *Fast sparse matrix-vector multiplication by exploiting variable block structure*, in High Performance Computing and Communications, Springer, 2005, pp. 807–816.

[8] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel, *Optimization of sparse matrix-vector multiplication on emerging multicore platforms*, Parallel Computing, 35 (2009), pp. 178–194.

A THREAD-SCALABLE PERFORMANCE PORTABLE UNORDERED MAP FOR MANYCORE ARCHITECTURES *

PAUL R. ELLER† AND H. CARTER EDWARDS‡

Abstract. An unordered map (*a.k.a.,* hash map) is a commonly used container for performant storage and retrieval of key-value pairs. High performance computing (HPC) algorithms on modern shared-memory many-core architectures require that an unordered map be *thread scalable*; that concurrent insert, delete, and find operations by thousands of threads are performant. Furthermore, these algorithms and the supporting unordered map must be performance portable across diverse many-core architectures; *e.g.*, Intel Xeon Phi and NVIDIA GPU.

Kokkos is a programming model and C++ library that enables applications and domain libraries to develop performance portable C++ code. To better support these codes a collection of commonly used containers, such as the unordered map, are being developed on the Kokkos library. These containers deviate from the classic C++ standard containers in order to meet thread-scalability requirements. This paper documents the design and performance analysis of Kokkos' performance portable and thread-scalable unordered map.

1. Introduction. Unordered maps are associative containers that store key-value pairs and provide fast access to individual elements of the map. Key values are used to uniquely identify each element, while the mapped value contains the data associated with the key. Internally the elements are not sorted in any particular order, but are instead organized into buckets. Each element is inserted into a bucket based on the hash of its key, providing fast access to individual elements. Unordered maps provide operations such as insert, find, and delete to modify or retrieve information from the map.

There are many unordered map implementations for CPUs that perform well in serial. However the number of cores per processor has increased in recent years. More importantly co-processors such as graphics processing units (GPUs) and Xeon Phi co-processors have been developed that provide hundreds or thousands of cores per processor. These processors often require multiple threads per core to obtain good performance, resulting in the need for algorithms that can effectively take advantage of thousands of threads per processor. Therefore we need unordered map algorithms that are capable of performing using thousands of threads in parallel.

In order to obtain good parallel performance on many-core devices, the algorithms must be both thread-safe and thread-scalable. Thread-safe means that multiple threads must be able to operate on the container without causing race conditions that can harm the integrity of the data. Thread-scalable means that the multiple threads must be able to operate on the container efficiently. Many unordered map implementations do not provide thread-safety, and the ones that do may not scale to the hundreds or thousands of threads needed to obtain good performance on many-core architectures.

1.1. Kokkos. Kokkos [9, 8, 10, 11] is a C++ library and programming model for expressing data parallelism on manycore architectures. Kokkos provides performance portable parallel algorithms (for, scan, and reduce), atomics, and basic thread scalable containers. This allows users to write code that can be compiled without modification and run on multiple different many-core architectures. This minimizes the amount of architecture specific knowledge that users must possess, while allowing Kokkos to provide architecture specific

*This is an early version of a paper that will be published in a journal after some additional work is completed.

†Department of Computer Science, Univerity of Illinois at Urbana-Champaign, eller3@illinois.edu

‡Sandia National Laboratories, hcedwar@sandia.gov

tuning. Kokkos attempts to provide portable user code that performs as well as architecture specific code while sticking to a small, straight-forward API.

1.2. Kokkos Unordered Map. Kokkos provides users with access to a thread-safe and thread-scalable unordered map that runs efficiently on a wide variety of many-core architectures as well as traditional CPUs. This unordered map is designed as a hash table with separate chaining with operations to insert, find, and erase key-value pairs. The unordered map is designed based on the C++ standard unordered map. However we restrict users to either insert or delete key-value pairs at a given time and prevent memory from being reallocated within a parallel operation in order to ensure that we obtain high performance on many-core architectures. This allows us to implement the map as a wait-free linked list and avoid dealing with additional complications to the data structure that limit performance.

2. Related Work. There are a number of hash map algorithms that have been developed, but only a few of these algorithms are thread-safe and capable of running on many-core architectures. Few of these algorithms have been implemented in publically available libraries. The best available implementations we were able to find were in the CUDA Data Parallel Primitives Library, the Intel Threaded Building Blocks template library, and the MultiThreaded Graph Library.

2.1. Non-threaded Hash Maps. A number of CPU based hash maps exist. The C++ standard library provides an unordered map, but this map is not designed to be thread-safe and thread-scalable. The Boost library also provides an unordered map, but once again this is not a thread-safe and thread-scalable implementation. Java provides a ConcurrentHashMap that provides hash map functionality without requiring the user to synchronize in multi-threaded applications. However there does not appear to be a simple approach for running java code on many-core architectures.

2.2. CUDPP. The CUDA Data Parallel Primitives Library [17] is a library of data-parallel algorithm primitives for GPUs written with CUDA. These primitives are used as building blocks for a wide variety of data-parallel algorithms. This library includes an implementation of hash tables based on the work of Alcantara et al. [2, 3]. CUDPP provides an efficient implementation of this hash table, but the functionality of this table is limited.

Building the hash table requires a list of all keys and values. Once the hash table has been built, CUDPP does not provide the functionality to add or remove keys from the hash table. As a result, CUDPP is able to make additional optimizations to gain improved performance, but only for certain use cases. This does not meet the needs of many of the applications that we need hash tables for, so we do not further investigate CUDPP.

2.3. Intel Threaded Building Blocks. Intel Threaded Building Blocks (TBB) [21, 19] provides a concurrent hash map implementation capable of running on CPUs and Xeon Phis. This hash map provides the ability to insert, find, and erase keys in the hash map. This provides functionality similar to that of Kokkos, although TBB does not run on GPUs.

2.4. MultiThreaded Graph Library. The MultiThreaded Graph Library provides a scalable hash map [15, 16] capable of running on CPUs. This hash map was in particular developed to run on a Cray XMT, a massively multithreaded supercomputer. This hash map uses a linear probing approach and provides the ability to insert, find, and erase keys in the hash map. This provides functionality similar to that of Kokkos, although this hash map does not run on GPUs.

2.5. Other Threaded Hash Maps. A few other papers investigate hash tables and related algorithms on many-core architectures, but they do not provide publically available

implementations of their algorithms. Moazeni and Sarrafzadeh [20] presents a compare-and-swap based lock-free hash table on GPUs that is based on chaining and compares it to cpu-based OpenMP hash tables and GPU lock-based hash tables, showing significantly improved performance. Heller et al [18] presents a lazy list-based implementation of a concurrent set object based on an optimistic locking scheme for inserts and removes.

3. Design and Algorithms. Kokkos' unordered map is designed as a hash table with separate chaining; each hash cell is the head of a linked list for the key-value pair associated with the key's hash value. Three operations are required: (1) inserting key-value pairs into the container, (2) finding key-value pairs within the container, and (3) erasing key-value pairs from the container. The design of these operations is challenged by requirements for performance *and* thread scalability. To satisfy these requirements the design of the Kokkos unordered map deviates from the C++ standard unordered map in two significant ways.

First, a parallel algorithm is restricted to either insert or delete key-value pairs, but not both. This restriction is enforced through an *insertable* state; when in the insertable state insert operations are permitted and erase operations are not. This design decision allows the insertion algorithm to use a wait-free linked list [13, 14] and avoid dealing with the "ABA" concurrency race condition that would complicate the underlying data structure and algorithms, and severely degrade performance. In our experience a particular HPC algorithm performs either insert or delete operations, but not both. We assume that this pattern will continue, or algorithms can be reasonably redesigned to use separate insert and delete phases.

Second, memory may not be reallocated from within a parallel operation. Thus an algorithm must set the capacity of the unordered map before performing a parallel operation, count insertion failures due to insufficient memory, reset the capacity after the parallel operation completes, and then repeat the parallel operation. In our experience the performance loss due to increased complexity of data structure, continually checking for insufficient memory, and reallocating memory from within a parallel operation is larger that the time required to repeat a simple parallel operation. We assume that this pattern will continue, or algorithms can be reasonably redesigned to separate the unordered map insertion portion of the algorithm into its own parallel operation.

Third, the key and value types must be trivially copyable.

3.1. Array-based Data Structure. The unordered map design uses a collection of arrays to store keys and values, maintain linked lists, and track claimed array entries. The array design is reflected in the index-based interface, in contrast to the standard C++ unordered map iterator-based interface. This design enables references (*a.k.a.*, indices) to key-value pairs to be independent of the memory space in which an unordered map resides, and allows an unordered map to be efficiently copied between memory spaces without invalidating references. The array-based design also enables optimizations associated with accessing contiguous arrays.

References to key-value pairs are simple integer index values, as opposed to a C++ iterator type. Three access functions accept an index value: (1) **exists(i)**, (2) **key_at(i)**, and (3) **value_at(i)**. The **exists** function returns whether a key-value entry exists for the input index. If an entry exists then **key_at** returns the key and **value_at** returns a reference to the value. If the unordered map is not constant the referenced value is modifiable.

Unordered map arrays are pre-allocated to a specified capacity. Subsequent insert operations claim array entries from this "pool" and erase operations release array entries back into this "pool." Claim and release operations are performed by atomically setting and clearing bits in a bitset.

3.2. Find Algorithm. The find algorithm (Alg 10) searches the unordered map for an input key and returns the array index i associated with that key. This is the index into the unordered map's key and value arrays: $Keys\,[i]$ and $Values\,[i]$.

Algorithm 10 Unordered Map Find

1: **define:** $E \equiv$ end-of-list marker
2: **input:** key
3: $i \leftarrow Start\,[hash\,(key)\ \ mod\ \#Start]$
4: **while** $i \neq E$ and $Keys\,[i] \neq key$ **do**
5: $\quad i \leftarrow Next\,[i]$
6: return i

The unordered map may be instantiated with the *hash* function used on line 3. The default function is a 32-bit MurmurHash3 [4]. The hash value, modulo the cardinality of linked list *Start* array, provides the starting location of the linked list associated with the input key (line 3).

The linked list is implemented by a set of entries in the *Next* array terminated by an end-of-list marker. The find algorithm iterates this list on lines 4-6 searching for an entry in the *Keys* array matching the input key. If a match is found the entry's index is returned, otherwise the end-of-list marker is returned.

3.3. Insert Algorithm. The insert algorithm (Alg 11) is designed to be performant on GPU architectures where in the innermost group (*e.g.*, CUDA warp) threads execute in "single instruction multiple thread" (SIMT). While CPU architectures do not have this SIMT performance requirement the algorithm is portable and performant on CPU architectures as well.

The insert algorithm has one of three results. (1) The insert of an input key-value pair succeeds. (2) An existing key-value pair is found with an equal key so the input value was not set. (3) The insert fails due to insufficient capacity. A user should count failures within a parallel reduction algorithm, resize the unordered map to the required capacity, and repeat the parallel algorithm. A successful insert results in an update of the linked list associated with the hashed key.

When the first entry is added to a linked list the *Start* array entry is updated. When a subsequent entry is added a *Next* array entry is updated. The memory address of the array entry to be updated is maintained in the address variable (*a.k.a.*, pointer) a (line 2 and 7).

3.3.1. Search for Input Key. Next the linked list is searched for the input key (lines 6-9). The linked list may be appended during this iteration so all reads of the *Keys* and *Next* arrays are marked *volatile* to force reads from shared, global memory. If the key is found the index of this entry will be returned with an *existing* status. If during a previous iteration an entry in the arrays was claimed then this entry is released (line 12). If the iteration search fails to find the input key then an attempt is made to append the linked list with the input key and value.

3.3.2. Insert New Key. First an unused array entry must be claimed (line 17). The claim algorithm searches a bitset for an unclaimed entry, claims that entry by atomically setting an associated bit in the bitset, and if the atomic-set operation fails it will continue the search. If an unused entry cannot be found a *failed* status is returned (line 19).

When an entry is successfully claimed the *Keys* and *Values* array entries are set. A memory-fence is applied to insure that these array entries are written in global memory before attempting to update the linked list. This memory-fence guarantees that the *Keys* entry queried on line 6 corresponds to the linked list entry queried on line 7.

Algorithm 11 Unordered Map Insert

```
1: input: key, value
2: a ← address_of (Start [hash (key) mod #Start])
3: j ← E
4: while ! result do
5:     i ← at (a)
6:     while i ≠ E and Keys [i] ≠ key do
7:         a ← address_of (Next [i])
8:         i ← at_address (a)
9:     if i ≠ E then
10:        if j ≠ E then
11:            release(j)
12:        result ← (existing, i)
13:    else
14:        if j == E then
15:            j ← claim()
16:            if j = E then
17:                result ← (failed, E)
18:            else
19:                Values [j] ← value
20:                Keys [j] ← key                                  ▷ memory fence
21:        if j ≠ E and CAS (a, E, j) then
22:            result ← (success, j)
23: return result
```

The linked list is updated with an atomic compare-and-swap (CAS on line 25). If this update is successful the index of this new entry is returned with a *success* status. The CAS update will fail when another thread appends the linked list between this thread's search (lines 6-9) and attempt to append. A CAS failure could be due to the same or different key appended to the linked list. In either case the linked list search iteration must resume.

3.3.3. Hashing Quality. The linked list is exclusively appended while insert operations are permitted. This design decision allows the linked list search iteration to resume at the previous location and only search new entries that have been appended since the prior search. We assume that the hashing function will distribute the set of inserted keys such that the cardinality of any given linked list will be relatively small compared to the capacity of the unordered map. If this "hashing quality" assumption does not hold then performance will degrade with the cardinality of the linked lists and frequency of searching long linked lists.

An entry is claimed only once per insert operation (line 17). This entry is held until the insert operation succeeds (line 24) or an existing entry is found (line 8) to avoid repeated claim and release operations. If an entry is claimed and subsequently the same input key is appended by another thread then the entry is released (line 12). Even with good hashing quality the claim function requires an increasing effort to find an unused array entry. As such the best performance is obtained with both good hashing quality and keeping the unordered map no more than 85% filled (*a.k.a.*, 85% density). While insert operation are unlikely to fail even up to 95% filled, performance typically degrades beyond 85% filled.

3.3.4. SIMT Performance. When a SIMT group concurrently calls the insert operation their execution will be "lock step." The insert algorithm's iterations and branches are carefully designed to avoid potential SIMT deadlock conditions. Furthermore, iterations are

designed so that thread members of a SIMT group will either execute iterations relevant to the thread's progress through the algorithm, or will complete the algorithm and thus be masked out of the SIMT group's iterations.

3.4. Erase Algorithms. The erase algorithm has two distinct phases: (1) marking entries to be removed from the unordered map and then (2) removing marked entries from the linked list. Marking entries to be removed is simply finding the entry and releasing its corresponding bit in the bitset. Since erase and insert operations are not permitted to occur within the same parallel operation releasing the entry will not conflict with the insert algorithm.

Algorithm 12 Unordered Map Erase Marking

```
1: input: key
2: i ← find(key)
3: if i ≠ E then
4:     release(i)
5: return i ≠ E
```

The removing phase of the erase algorithm (Alg 13) executes a parallel loop over all potential linked lists. If a member has been marked as released in the bitset then it is removed from the linked list.

Algorithm 13 Unordered Map Erase Removing

```
1: parallel for all i ∈ [0..#Start) do
2:     j ← Start[i]
3:     ▷ Remove entries until a new start is found
4:     while j ≠ E and ! Bits[j] do
5:         n ← Next[j]
6:         Next[j] ← E
7:         Start[i] ← n
8:         j ← n
9:     if j ≠ E and Bits[j] then
10:        ▷ Remove remaining entries
11:        p ← j
12:        j ← Next[p]
13:        while j ≠ E do
14:            n ← Next[j]
15:            if Bits[j] then
16:                p ← j
17:            else
18:                Next[p] ← n
19:                Next[j] ← E
20:            j ← n
21: end for
```

4. Performance Analysis.

4.1. Test Setup. Performance tests are run on the Compton and Shannon test bed clusters located at Sandia National Lab. Test bed configurations are shown in figure 4.1. Compton contains nodes with dual socket Intel Xeon CPUs and Intel Xeon Phi co-processors. Shannon contains nodes with dual socket Intel Xeon CPUs and Nvidia Kepler K40M GPUs.

Table 4.1: Configurations of test bed clusters.

Name	Compton	Shannon
CPU	2x Intel E5-2670 2.6GHz HT-on	2x Intel E5-2670 2.6 GHz HT-off
Co-Processor	Intel Xeon Phi 57c 1.1GHz	Nvidia K40m ECC on
Memory	24 GB	128 GB
CPU Threads	32	32
Device Cores	57 (228 Threads)	2880
OS	RedHat 6.1	RedHat 6.5
Compiler	ICC 15.0.1	GCC 4.7.2 + CUDA 7.0.28

Each test is performed using a single device, where device refers to a dual socket Xeon node, a single Xeon Phi, and a single Nvidia Kepler GPU respectively.

We compare the performance of Kokkos on Intel Xeon CPUs using OpenMP [22], Nvidia Kepler GPUs using CUDA [1], and Intel Xeon Phis using OpenMP. We were not able to find any comparable unordered map implementations on GPUs to compare with Kokkos. Kokkos is compared against TBB on Xeon Phis. We look at the number of insert, find, and erase operations we are able to perform per microsecond as we increase the number of keys used in each test in order to more clearly understand the cost of each operation.

First we run throughput performance tests to better understand how these maps are able to perform under fairly straightforward circumstances. We use these tests to evaluate the ability of the unordered maps to efficiently insert, find, and erase key-value pairs under a variety of circumstances. Each of these tests uses a parallel_for operation on key counts ranging from 10 thousand to 82 million. All tests use 32-bit unsigned integers for the key and one or more 32-bit unsigned integers for the value. We experiment with the following parameters:

- Capacity: The number of key-value pairs that the unordered map is initialized to hold. We fix the density of the map and create unordered maps that have low (115%), medium (172%), and high (230%) capacities compared to the number of keys we plan to insert. This allows us to observe how the unordered map performs when there are varying amounts of free space in the map.
- Density: The percentage of the unordered map that is filled with keys. We fix the capacity of the map and increase the number of keys in the map up to the capacity. This allows us to observe how the unordered map performs as the number of empty elements in the map decreases and the number of conflicts increases.
- Repeats: The number of times a particular key value is repeated. We insert, find, or erase each key from 1 to 8 times. This allows us to see how the performance changes when some keys are used multiple times, in particular when multiple threads may access the map with the same or similar keys.
- Value Size: The memory size of the value in the key-value pair. We vary the value size from 1 to 32 unsigned integers. This allows us to observe how performance changes as the map contains more data.
- Key Pattern: The pattern of keys used by threads to access the map. We explore three key patterns:
  - Sequential - Each thread uses keys in increasing order (0,1,2,3,...)
  - Strided - Each thread uses keys separated by a stride (0,16,32,48,...)
  - Random - We create a sequential array of keys and then randomly permute the array. Each thread accesses this array in sequential order to obtain a random

key.

This allows us to observe how performance changes as threads use keys that are nearby or far apart.

Algorithm 14 shows the Kokkos insert throughput performance test. Similar performance tests are used for find and erase. For each test we set the default case as using the sequential key pattern, no repeats, a value size of one, and a capacity about 15% larger than the number of key-value pairs. We execute each performance test ten times and report the fastest run time for each test.

4.2. Similarity of Find and Erase. The find and erase performance tests show fairly similar patterns throughout each test. This is expected since erase is implemented using find. Erase is slightly slower than find due to having to perform some extra operations to remove each element from the map. For simplicity we only focus on the performance of insert and find in most of the following results.

Algorithm 14 Throughput Insert Performance Test

```
define: keytype ≡ unsigned int
define: valuetype ≡ array of size VSIZE
define: maptype ≡ map(keytype, valuetype)
map ← maptype(maxkey * capacity)
Start timer
parallel for all i ∈ [0..N) do
    value ← valuetype(VSIZE)
    if test = sequential then
        INSERT(i/repeat,value)
    if test = strided then
        INSERT(i % N / repeat,value)
    if test = shuffled then
        INSERT(random_values(i),value)
end for
Fence
Stop timer
```

4.3. Comparison with TBB. We first run performance tests comparing the performance of Kokkos and Intel Threaded Building Blocks (version 4.3 update 5) on Xeon Phis, allowing us to better understand how these unordered map implementations compare.

Figure 4.1 clearly shows that Kokkos outperforms TBB on Xeon Phis for the default case. Kokkos in particular excels at inserting and erasing key-value pairs, significantly outperforming TBB. Kokkos also outperforms TBB at finding values in the map, although TBB performs fairly well, approaching the performance of Kokkos at times. Additionally we can see that the average performance of TBB can be much lower than the best case performance, demonstrating that there can be significant variation in runtimes for TBB. Meanwhile Kokkos gives very consistent performance, achieving average runtimes that are very similar to the best case performance. Additional tests for capacity, density, repeats, value size, and key pattern also showed Kokkos outperforming TBB. Since we can clearly see that Kokkos performs better on Xeon Phis than TBB, we will focus only on Kokkos for the remaining performance tests.

4.4. Throughput Performance Tests.

4.4.1. Capacity. We experiment with fixing the density of the map and experimenting with different capacity sizes to show the impact of having extra space in the map, making

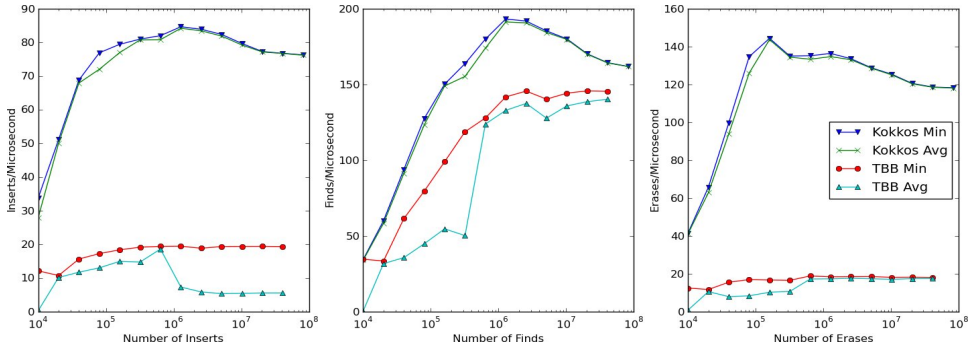


Fig. 4.1: Comparison of Kokkos and TBB minimum and average performance for 10 tests on the Xeon Phi for the insert, find, and erase operations for the default case (115% capacity, fixed density, no repeats, value size of 1, and sequential key pattern).

it less likely that multiple keys will hash to the same array index. We would expect having some extra capacity to improve performance due to having fewer keys mapping to the same array index.

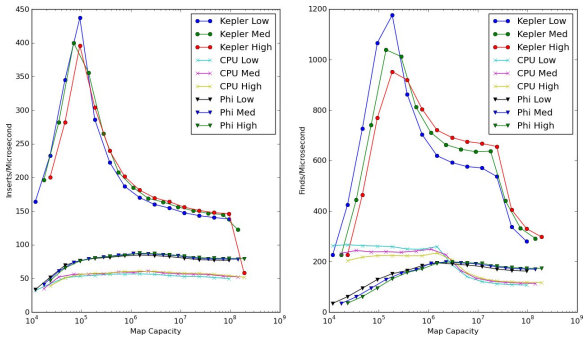


Fig. 4.2: Kokkos insert and find performance for unordered maps with low (115%), medium (172%), and high (230%) capacities.

Figure 4.2 shows the insert and find performance for low, medium, and high capacities as we increase the number of keys. Inserting values on the Xeon CPUs and Xeon Phis maintained fairly consistent performance as we increase the capacity. Both devices show some slight performance benefit due to using higher capacities. The Kepler GPU shows improved performance for lower capacities at smaller key counts and higher capacities at larger key counts. We also see that for the largest key counts, there is a drop in performance on the GPU. Additional tests demonstrated that once the initial capacity becomes large enough, there is a performance drop off regardless of the number of keys inserted into the map.

When finding keys, all three devices showed improved performance at smaller key counts for low capacities, but slight better performance at larger key counts for higher capacities. The Xeon CPU and Xeon Phi both showed more consistent performance for all key counts, while the Kepler GPU showed more variable performance. When using texture memory for the largest test for on the Kepler GPU, we ran out of out of memory and instead use the run time from the test without texture memory.

We note that the peak performance for the insert and find operations on the Kepler GPU occur when data fits into the 1.5MB L2 cache on the GPU. Once data no longer fits into L2 cache, we see decreased performance. Similarly for the find operation on the Xeon CPU we see decreased performance once data no longer fits into the 20MB cache. The insert operation is not able to take advantage of the cache due to the data structures being continuously modified, causing each thread to have to read data from main memory. Therefore the insert operation performance on the Xeon CPU maintains fairly constant performance.

4.4.2. Density. We experiment with fixing the capacity of the map and then using an increasing number of keys. This allows us to better understand the performance as the map density increases. We would expect performance to become worse as the density increases due to the increased likelihood that multiple keys will map to the same array index.

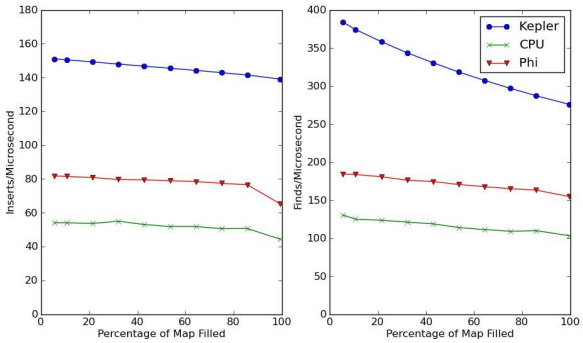


Fig. 4.3: Kokkos insert and find performance for unordered maps with an initial capacity of about 95 million for up to 95 millions key-value pairs. We increase the number of key-value pairs in the map to vary the density of the map. We use no repeats, a value size of 1, and the sequential key pattern.

Figure 4.3 shows that the insert and find performance for all three devices slowly decreases as we approach full capacity. However we do not see a large drop off in performance as we near full capacity, instead only seeing slight performance degradation on each device. We see a slightly larger drop in performance for the 99% capacity test, although this decrease is still fairly small. We also note that some inserts failed for the 99% capacity test. This shows that the unordered map maintains good performance for different map densities, although it may fail to insert all values if the number of key-value pairs approaches the capacity of the map.

4.4.3. Repeats. We experiment with using keys that are repeated multiple times. Repeating keys tests the ability of each device to perform well when multiple threads may be calling functions with the same or nearby keys. We expect insert and erase performance to improve as we have more repeats due to finding that the key-value pair they are looking for has already been inserted or erased, allowing the thread to return early. We expect find performance to improve due to the increased likelyhood that a key-value pair resides in cache due to that key-value pair or a nearby key-value pair having already been found.

Figures 4.4 and 4.5 show the impact of using repeated key values on inserting keys into the map and finding keys in the map. As expected, the performance increases for all devices as we increase the number of repeated keys. We see fairly similar performance for each number of repeats for smaller key counts, but for larger key counts we see more significant

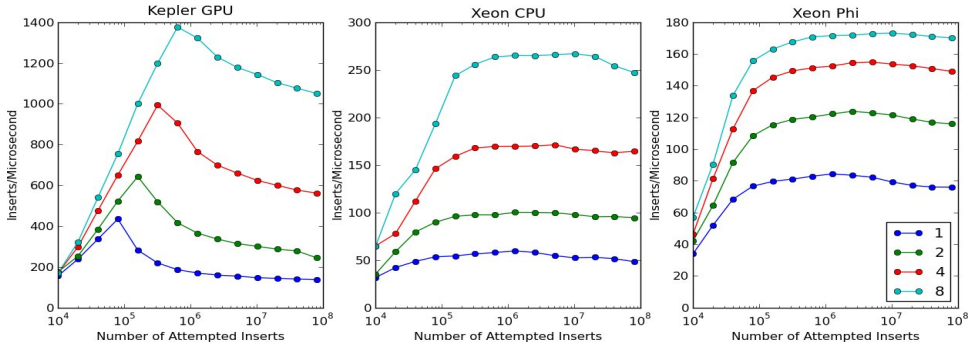


Fig. 4.4: Kokkos insert performance with key values repeated 1, 2, 4, or 8 times on the Kepler GPU (up to 1376 inserts/microsecond), Xeon CPU (up to 267 inserts/microsecond), and Xeon Phi (up to 173 inserts/microsecond).

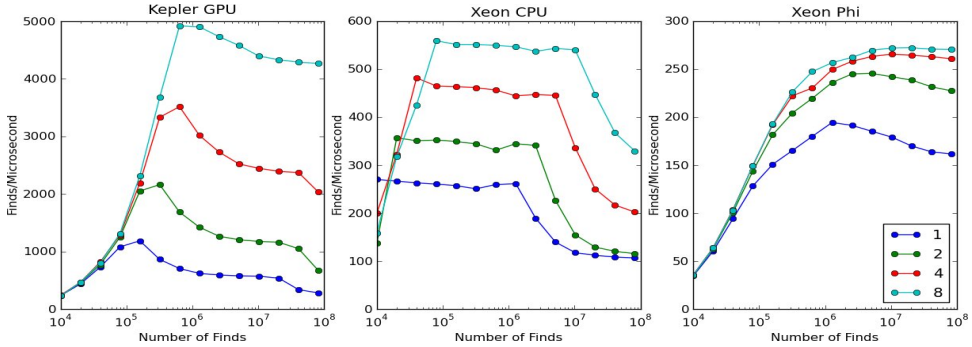


Fig. 4.5: Kokkos find performance with key values repeated 1, 2, 4, or 8 times on the Kepler GPU (up to 4923 finds/microsecond), Xeon CPU (up to 559 finds/microsecond), and Xeon Phi (up to 272 finds/microsecond).

performance improvements as we increase the number of repeats.

4.4.4. Value Size. Tests experimenting with using values of increasing size are used to show the ability of these functions to perform well for key-value pairs that have values that require more memory. We expect the insert tests to run more slowly with larger value sizes due to the cost of accessing larger amounts of memory, while we expect the find tests to have similar performance for each value size since they do not require us to access the value. We perform tests with value sizes ranging from 1 to 32 unsigned integers on each device.

Figure 4.6 shows the performance of inserting and finding key-value pairs with larger values on each device. Inserting larger values on the Xeon Phi does not show a noticeable difference in performance. On the Xeon CPU, we see a slight performance decrease as we increase the size of the value. On both the Xeon CPU and Xeon Phi we see fairly consistent performance as we increase the number of keys for all value sizes tested. On the Kepler GPU however, we see a more significant performance impact as we increase the value size. Larger value sizes produce a signficiant decrease in performance on the Kepler GPU. In particular we note that the performance on the Kepler GPU decreases below the performance on both the Xeon CPU and Xeon Phi as we increase the size of the value and increase the number

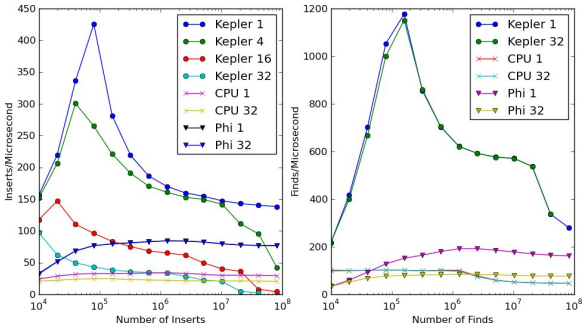


Fig. 4.6: Kokkos insert and find performance for key-value pairs with values ranging from 1 to 32 unsigned integers.

of keys.

Finding larger values did not produce a noticeable difference in performance in almost all cases. On the Kepler GPU and Xeon CPU we did not see a noticeable difference in performance as we increased the value size. On the Xeon Phi we also see consistent performance on value sizes from 1 to 16, however once the value size increased to 32 we saw a decrease in performance.

4.4.5. Key Pattern. We experiment with using sequential, strided, and randomly shuffled key insert, find, and erase patterns with the unorderd map. These patterns test the ability of each device to perform well when each thread is using keys that are nearby, far apart, and randomly selected. We would expect the map to perform better when each thread is using keys that are far apart. The key pattern that produces this behavior depends on the device, as each device will map array indices to threads in a different manner. Additionally testing both the sequential and strided key patterns is more likely to cause conflicts as multiple threads try to operate on the same or nearby key values. This can demonstrate that sequential or strided key patterns may be more effective on different devices.

Figure 4.7 shows that on each device, the sequential, strided, and randomly shuffled tests produced fairly similar performance for each key count. We see slightly worse performance on the Kepler GPU for inserting randomly shuffled keys on lower key counts. We also see slightly decreased insert performance on the Xeon Phi for shuffled and strided key patterns. We see similar patterns for finding keys as we do for inserting keys on the Kepler GPU and Xeon Phi. However we see more varied performance on lower key counts on the Xeon CPU.

However if we look at the performance for different key patterns with repeats, we see that the key pattern can make a large difference. Figure 4.8 shows that the sequential insert pattern outperforms the strided insert pattern for all three devices when we use 8 repeats. On the Kepler GPU we see that for smaller test sizes both key patterns perform well, but the strided performance decreases for larger key counts.

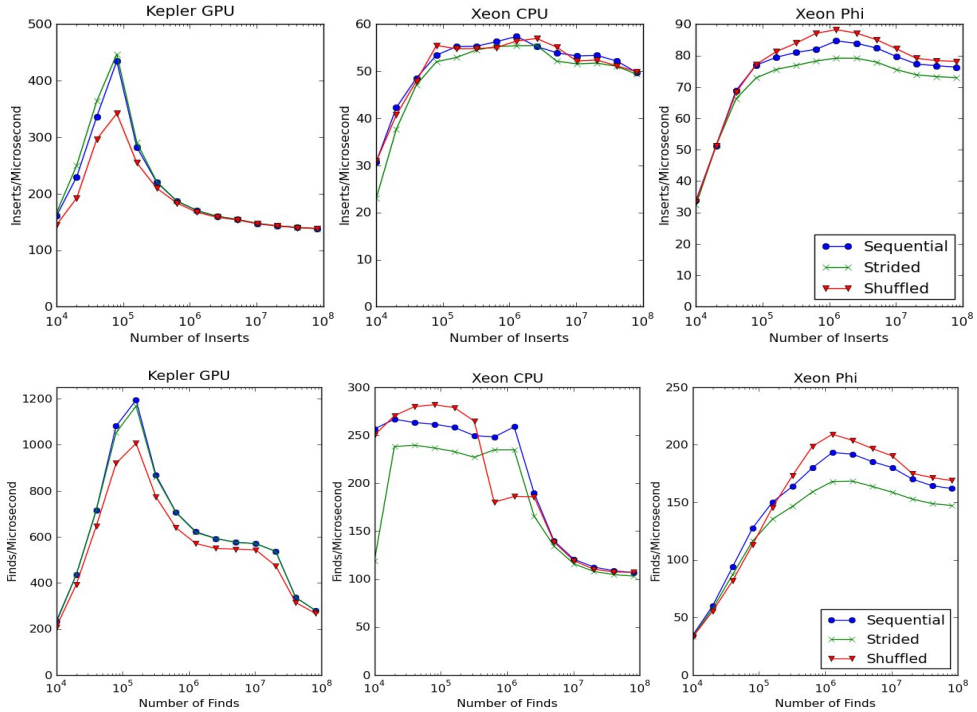


Fig. 4.7: Kokkos unordered map performance for sequential (Seq), strided (Str), and randomly shuffled (Shuf) key insert and find patterns on the Kepler GPU (up to 446 inserts and 1194 finds per microsecond), Xeon CPU (up to 57 inserts and 281 finds per microsecond), and Xeon Phi (up to 88 inserts and 209 finds per microsecond).

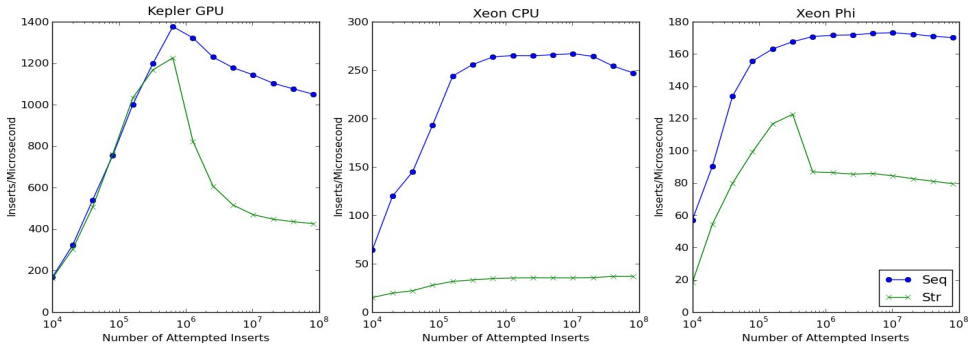


Fig. 4.8: Kokkos unordered map performance for sequential (Seq), strided (Str), and randomly shuffled (Shuf) key insert patterns with 8 repeats on the Kepler GPU (up to 1376 inserts/microsecond), Xeon CPU (up to 267 inserts/microsecond), and Xeon Phi (up to 173 inserts/microsecond).

5. Performance Evaluation with Mini-Applications. Next we experiment with two mini-applications based on real-world applications that unordered maps are commonly used for. The first mini-application is a non-linear finite element code (FENL) and the second is a graph generation mini-application. FENL is a mini-application that uses the finite element method to solve a nonlinear problem with Newton iterations. We focus on the performance of contructing and filling a sparse linear system using the Kokkos unordered map.

The second mini-application is a graph generation algorithm from the MultiThreaded Graph Library. We run tests to randomly generate a R-MAT graph and an Erdos-Renyi graph. This algorithm is implemented using an unordered map by inserting each generated edge into the map to ensure that we generate a full set of unique edges.

5.1. Nonlinear Finite Element Mini-Application. We use the nonlinear finite element (FENL) mini-application [7] to test our unordered map in a more realistic application. This mini-application uses the finite element method to solve a nonlinear problem using Newton iterations. FENL solves the simple scalar nonlinear equation $-k\Delta T + T^2 = 0$ on a 3-d box domain. The geometry and boundary conditions are restricted in order to obtain an analytic solution that we can use to verify correctness. This mini-application uses linear hexahedral finite elements with 2x2x2 numerical integration with non-affine mapping of vertices for non-uniform element geometries. The resulting linear system is solved with a conjugate gradient iterative solver.

We focus on constructing and filling the sparse linear system using the Kokkos unordered map in order to demonstrate that Kokkos provides a thread safe, thread scalable, performant unordered map implementation. We evaluate the performance of this map on the Xeon CPU, Kepler GPU, and Xeon Phi.

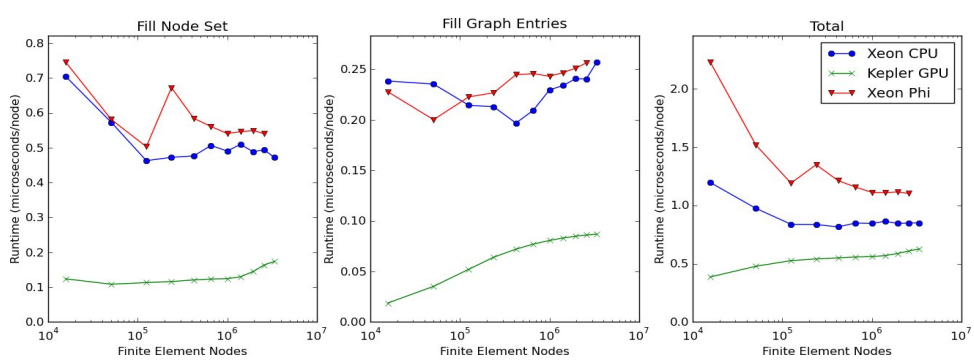


Fig. 5.1: Nonlinear Finite Element (FENL) performance for fill node set and fill graph entries routines using Kokkos unordered map and the total performance for generating an element graph map.

The main FENL routine constructs a sparse graph from an unordered map. These routines fill the unordered map with elements, compute the row-offset, allocate the CRS graph structure, fill the CRS graph, and then sort the column entires in each CRS graph row. Once the element graph map is constructed, we can create the sparse matrix from the graph. We can then finish setting the relevant boundary conditions and then perform nonlinear Newton iterations, using a conjugate gradient iterative solver at each nonlinear iteration, to reach the solution.

A number of routines are used to construct the sparse graph, but we focus on the two routines that use the unordered map. The fill map operator fills the unordered map with

elements and the fill graph operator then processes the unordered map to generate the CRS graph entries. These routines do not need to associate a value with the key.

The fill map operator fills the map with elements by looping over row-node and column-node pairs and inserting locally owned pairs into the unordered map. If the key is successfully inserted into the map, then we use an atomic fetch add to update the row or column node. We may have to repeat this routine a second time if the initial capacity was too low as discussed in section 3.3.

We fill the CSR graph using the unordered map that was generated by the fill map operator. This routine requires us to iterate over the contents of the unordered map. As discussed in section 3.1, we treat the unordered map as an array by looping over the contents of the unordered map and checking if an index is in the unordered map. If the index is valid, we get the key at that index, compute the offset, and add the node to the list of graph entries.

The FENL mini-application is run multiple times on problem sizes ranging from $25x25x25$ (15 thousand) elements to 150x150x150 (3.375 million) elements. The test results for FENL are shown in figure 5.1. These results show fairly consistent performance per node on the Kepler GPU and Xeon CPU as we increase the number of finite element nodes. On the Xeon Phi we see some decreases in performance per node at times, but overall we see performance improve as we use more finite element nodes. Overall this demonstrates effective performance using Kokkos on multiple devices for a realistic nonlinear finite element problem.

5.2. Graph Generation Mini-Application. We use graph mini-applications from the MultiThreaded Graph Library (MTGL) [5] to test Kokkos. MTGL is a collection of algorithms and data structures designed to run on shared-memory platforms or on multi-core systems. The software and API is modeled after the Boost Graph Library, with modifications to take advantage of shared memory machines.

We adapt part of MTGL to run using Kokkos on many-core architectures. We run R-MAT and Erdos-Renyi graph generation tests to generate graphs with m unique edges. Each graph generation test uses the same random graph generation routine but provides a unique edge generation routine. These graph generation routines may generate the same edge multiple times, resulting in a collision. These collisions means we need to generate extra edges to ensure that we have a graph containing m unique edges.

The R-MAT (recursive matrix) graph [6] recursively sub-divides the adjacency matrix of the graph into four equal-sized partitions and distributes the edges within these partitions with unequal probabilities. These unequal probabilities make it more likely that certain edges will be generated multiple times, resulting in a higher likelihood of collisions.

The Erdos-Renyi [12] graph has n vertices and generates two random numbers to select two vertices and create an edge between them. This results in an algorithm where there is an equal probability that there will be an edge between any two vertices, resulting in a smaller likelihood that the same edge will be generated multiple times.

This mini-application is implemented by inserting each edge into an unordered map. If the insert is successful then we know the edge is unique. We expect the performance of R-MAT graph generation to be limited by the cost of collisions, especially as the number of edges increases. We expect the Erdos-Renyi graph to have fewer collisions, allowing the unordered map to insert edges with a higher success rate and obtain better performance as we increase the graph size.

The primary MTGL mini-application random graph generation routine attempts to add edges to the map in phases, calling one device routine to insert many edges at once. A second series of routines is called once we have less than a full phase of edges to add to ensure that all edges are added to the map that would be added in serial and keep the graph deterministic.

The routine to insert many edges at once generates an edge, performs some simple checks to verify that the edge is not a duplicate, generates a key, and then attempts to insert the key into the map. If the key is successfully inserted into the the map, we add the key to a list of edges, using an atomic fetch add to ensure that we use a unique index. This routine does not need to associate a value with the key.

The routine to add the remaining edges uses a similar process, but takes extra steps to ensure that we are adding all of the edges that would have been added in serial in order to keep the graph deterministic. This requires generating a set of edges for the full phase, but only adding the first edges generated that are needed to result in m unique edges being added to the graph. Once we have generated a full list of edges, we can call a routine to generate the graph.

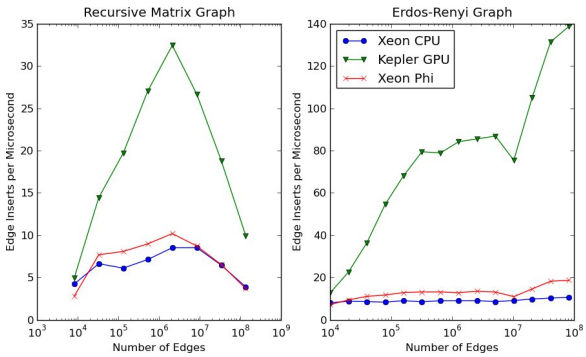


Fig. 5.2: Kokkos MTGL performance for generating a list of unique edges for a R-MAT graph (left) and Erdos-Renyi graph (right).

Figure 5.2 shows the performance of Kokkos for each device for the R-MAT and Erdos-Renyi graph generation mini-applications. We see that for the R-MAT graph generation routine the performance increases as we increase the number of edges to a point, but then performance drops off. For the Erdos-Renyi graph generation routine, on the Kepler GPU the performance increases as we increase the number of edges, while on the Xeon CPU and Xeon Phi the performance is very consistent for all edge counts. This behavior is expected as the R-MAT graph has many collisions as we increase the size of the graph. This results in more failed inserts and requires additional steps to generate and insert m unique edges. The Erdos-Renyi graph has few collisions, resulting in fewer failed inserts, allowing us to obtain much better performance as we increase the size of the graph.

6. Conclusions. Kokkos provides a thread-safe, thread-scalable unordered map (a.k.a., hash map) container capable of performing well on modern shared-memory many-core architectures. This container provides users with insert, delete, and find operations that are performant when called by thousands of threads at the same time.

While there are many unordered map implementations that have been developed, there are few implementations available that are both thread-safe and thread-scalable. Only Intel threaded building blocks provides similar functionality as the Kokkos unordered map. Unlike other implementations, the Kokkos unordered map is portable, allowing us to run on both traditional CPUs as well as many-core architectures such as GPUs and Xeon Phis.

Performance tests demonstrate that Kokkos clearly outperforms Intel threaded building blocks on Xeon Phis. We also see that the Kokkos unordered map is able to maintain high performance on each architecture when varying parameters such as the capacity and density of the map, the number of times a particular key value is repeated, the memory size of the

value, and the key pattern. Experiments using the Kokkos unordered map with the nonlinear finite element (FENL) and graph generation mini-applications demonstrates that the Kokkos unordered maps are able to provide good performance on realistic applications on a variety of architectures.

REFERENCES

[1] *Nvidia cuda zone*. https://developer.nvidia.com/cuda-zone, Aug. 2015.

[2] D. A. Alcantara, *Efficient Hash Tables on the GPU*, PhD thesis, University of California, Davis, 2011.

[3] D. A. Alcantara, A. Sharf, F. Abbasinejad, S. Sengupta, M. Mitzenmacher, J. D. Owens, and N. Amenta, *Real-time parallel hashing on the gpu*, ACM Transactions on Graphics, (2009), p. 154:1154:9.

[4] A. Appleby, *Murmarhash3*. code.google.com/p/smhasher, April 2011.

[5] J. W. Berry, B. Hendrickson, S. Kahan, and P. Konecny, *Software and algorithms for graph queries on multithreaded architectures*, Parallel and Distributed Processing Symposium, International, 0 (2007), p. 495.

[6] D. Chakrabarti, Y. Zhan, and C. Faloutsos, *R-mat: A recursive model for graph mining*, SIAM Data Mining, (2004).

[7] H. C. Edwards, *Minifenl: Fully hybrid parallel and performance portable nonlinear finite element mini-application using mpi+kokkos*, May 2014.

[8] H. C. Edwards and D. Sunderland, *Kokkos array performance-portable manycore programming model*, in PMAM, Feb. 2012, pp. 1–10.

[9] H. C. Edwards, D. Sunderland, C. Amsler, and S. Mish, *Multicore/gpgpu portable computational kernels via multidimensional arrays*, in Cluster Computing (CLUSTER), 2011 IEEE International Conference on, IEEE, Sept. 2011, pp. 363–370.

[10] H. C. Edwards, D. Sunderland, V. Porter, C. Amsler, and S. Mish, *Manycore performance-portability: Kokkos multidimensional array library*, Scientific Computing, (2012), pp. 89–114.

[11] H. C. Edwards, C. R. Trott, and D. Sunderland, *Kokkos: Enabling manycore performance portability through polymorphic memory access patterns*, Journal of Parallel and Distributed Computing, 74 (2014), pp. 3202–3216.

[12] P. Erdos and A. Renyi, *On random graphs. i*, Publicationes Mathematicae, 6 (1959), p. 290297.

[13] M. Fomitchev and E. Ruppert, *Lock-free linked lists and skip lists*, in the twenty-third annual ACM symposium, New York, New York, USA, 2004, ACM Press, pp. 50–59.

[14] H. Gao, J. F. Groote, and W. H. Hesselink, *Lock-free dynamic hash tables with open addressing*, arXiv.org, (2003).

[15] E. Goodman, D. Haglin, C. Scherrer, D. Chavarria-Miranda, J. Mogill, and J. Feo, *Hashing strategies for the cray xmt*, in Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on, April 2010, pp. 1–8.

[16] E. Goodman, M. N. Lemaster, and E. Jimenez, *Scalable hashing for shared memory supercomputers*, in Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, SC '11, New York, NY, USA, 2011, ACM, pp. 41:1–41:11.

[17] M. Harris, J. Owens, S. Sengupta, Y. Zhang, A. Davidson, R. Patel, and L. Wang, *Cuda data parallel primitives library documentation*. http://cudpp.github.io/, 2015.

[18] S. Heller, M. Herlihy, V. Luchangco, M. Moir, W. N. Scherer, and N. Shavit, *A Lazy Concurrent List-Based Set Algorithm*, in Principles of Distributed . . . , Springer Berlin Heidelberg, Berlin, Heidelberg, 2006, pp. 3–16.

[19] Intel, *Threaded building blocks*. https://www.threadingbuildingblocks.org/, 2015.

[20] M. Moazeni and M. Sarrafzadeh, *Lock-free hash table on graphics processors*, in Proceedings of the 2012 Symposium on Application Accelerators in High Performance Computing, SAAHPC '12, Washington, DC, USA, 2012, IEEE Computer Society, pp. 133–136.

[21] J. Reinders, *Intel Threading Building Blocks*, O'Reilly & Associates, Inc., Sebastopol, CA, USA, first ed., 2007.

[22] *The OpenMP API Specification for Parallel Programming*. openmp.org/, Aug. 2015.

CREATING AN AMGX ADAPTER WITHIN THE MueLu PACKAGE

E. FURST*, A. PROKOPENKO†, AND J. HU‡

Abstract. In the MueLu package, an adapter was created to allow the user to utilize the AmgX library as a preconditioner and solver in place of MueLu. MueLu is a software package that is part of Trilinos library implementing algebraic multigrid (AMG) algorithms. AmgX is a collection of linear solvers developed by NVIDIA that are executed on the GPU. The AmgX adapter was implemented to allow running AmgX on a single GPU device. A study was conducted to compare the performance of AmgX and MueLu libraries as preconditioners. Belos, a Trilinos software package implementing iterative linear solvers, was used to supply an outer iterative linear solver. MueLu was provided an input deck with either AmgX configuration settings or MueLu parameters. In both cases, the preconditioners were initialized with the matching settings. The comparison was run using Laplace operators of increasing problem sizes in two and three dimensions. It was found that on larger problem sizes, AmgX as a preconditioner produced faster times than MueLu.

1. Introduction. The desire to solve increasingly larger problems has made it necessary to find ways to better utilize computing resources. This is often accomplished through parallel implementations of algorithms and the use of distributed systems. Often, these large problems take the form of sparse linear systems. Multigrid algorithms are often used for the solution of these linear systems because of their algorithmic and parallel scalability [4].

In this work, we created an adapter in MueLu to give the user an option to use AmgX, an algebraic multigrid linear solver library developed by NVIDIA which allows for the solution of linear systems on graphics processing units (GPUs). Our goal was to implement this adapter so that it could be run both in serial and in parallel using the Message Passing Interface (MPI). We also wanted to make the adapter easy to use through the use of input decks. By having the AmgX options specified in an input deck, the adapter can be utilized with minimal if any changes to user code.

1.1. Multigrid Methods. Multigrid methods approximate the solution to a linear system through the creation and solution of coarser problems. The solutions to these coarser problems are combined to help accelerate the approximation of the original, finest problem. Approximate solvers referred to as pre-smoothers and post-smoothers help to quickly reduce certain error modes in the approximate solution on each level of the multigrid. Solvers used in smoothing are typically relaxation methods such as Jacobi or Gauss-Seidel. These relaxation methods are effective up until a point when only certain error modes remain. At this point the errors are called *smooth*, and it is beneficial to move to a coarser grid where these error modes appear differently and the relaxation method can be more effective. In order to move between levels interpolation and restriction operators are chosen. Interpolation matrices transfer solutions from coarse to fine levels, and restriction matrices restrict a residual from a fine level to a coarser level. Both MueLu and AmgX implement algebraic multigrid (AMG) methods, where these matrices are automatically generated. Thus, the hierarchy of grids in AMG is derived directly from the finest level using graph algorithms. Multigrid algorithms utilize various cycles to move from finer to coarser levels. For instance, a V-cycle involves one coarse grid correction step, and a W-cycle involves two coarse grid correction steps between the pre-smoother and the post-smoother. For a more comprehensive background on multigrid, see [4].

1.2. MueLu**.** The MueLu package is an algebraic multigrid library that exists within the Trilinos project. It provides a framework for the solution of large sparse linear systems

*College of Saint Benedict/University of Washington, eafurst@cs.washington.edu
†Sandia National Laboratories, aprokop@sandia.gov
‡Sandia National Laboratories, jhu@sandia.gov

using parallel multigrid preconditioning. MueLu is designed to be portable and efficient on many different architectures and relies upon the "MPI+X" paradigm where "X" can be threading or CUDA. MueLu can then exploit both distributed-memory parallelism and any of the various shared-memory parallel programming models.

This is accomplished through the use of the common interface of Kokkos, an implementation of a programming model for writing performance portable applications. Versatility is also a goal of the MueLu package, aiming to allow for the easy reuse and adaptability of algorithms to different problems. MueLu provides several multigrid algorithms including smoothed aggregation AMG [14], Petrov-Galerkin aggregation AMG [13], and energy-minimizing AMG [11]. MueLu gives users run-time control over the algorithms and parameters that it uses, through an XML or ParameterList input deck. MueLu can be used as a standalone multigrid solver or can be used as a multigrid preconditioner with Belos or AztecOO as a solver [12, 5].

MueLu uses several different Trilinos packages including Tpetra, Ifpack2, Belos, and Teuchos. Belos is a collection of standard iterative linear solvers and also contains flexible variants of CG and GMRES. It provides a linear solver developer framework and provides next-generation iterative linear solvers [3]. Tpetra is a package which implements linear algebra objects such as matrices and vectors. Tpetra is one of a few packages in Trilinos which implement linear algebra objects. Specifically, Tpetra is templated allowing objects to contain many different kinds of data including complex-valued types or floating-point types of varying precision. Also, because Tpetra uses Kokkos, Tpetra objects are easily portable to new and different computer architectures. Further, Tpetra supports "MPI+X" parallelism and distributed data [9]. The Teuchos package provides a set of common tools for Trilinos developers including reference counted pointers (RCPs), parameter lists, and XML parsers. Ifpack2 provides operators which can be used for multiple purposes including as preconditioners in iterative solvers or as smoothers for AMG. These operators include incomplete factorizations and relaxations [8].

2. AmgX: NVIDIA's Multigrid Linear Solver API. AmgX is a collection of multigrid linear solvers developed by NVIDIA. Various iterative linear solvers are implemented including BiCGStab, CG, GMRES, FGMRES, and Flexible CG. In addition, both classical algebraic multigrid and aggregation multigrid are implemented. Further, block Jacobi, multicolor Gauss-Seidel, multicolor DILU, multicolor ILU, and polynomial smoothers are implemented [10] These solvers run on the GPU and currently can be run MPI sequentially, utilizing one GPU device, or in MPI parallel, utilizing multiple devices. However, in this paper, we focus on the MPI serial usage. In addition, AmgX does support OpenMP applications, but it treats these applications as though they were single threaded applications and requires a *master* thread to be specified by the application for all communication with AmgX.

The AmgX API contains five core types of objects: *Config, Resources, Solver, Matrix, and Vector*. The *Config* object holds various parameter strings that correspond to the settings for the *Resources* and *Solver* objects. Settings for the *Config* object can be stored in a text file and uploaded or specified in a string. The configuration files contain a series of key-value pairs which specify various parameters and algorithms and is stored using the JSON file type. The *Resources* object contains information relevant to the resources that will be used by the AmgX library including information about GPUs and when running in parallel, information about the MPI communicator being used. The *Solver* object is created for the execution of algorithms to solve a linear system. AmgX allows for many different combinations of the algebraic multigrid methods and iterative linear solvers mentioned previously. The *Matrix* and *Vector* objects represent the sparse linear system to be solved and

can be stored either on the host or device. The *Matrix* and *Vector* elements can be either doubles or floats, and indices must be 32-bit integers. AmgX supports various matrix types and layouts including block matrices and distributed storage of matrices. However, AmgX only supports square matrices [10].

3. The AmgX Adapter. The AmgX adapter is currently only compatible with Tpetra objects. An AmgX solver object will be created if a `Tpetra::CrsMatrix` (Compressed Row Storage Matrix) is passed in along with AmgX configuration settings to the method

```
CreateTpetraPreconditioner(
  const Teuchos::RCP<Tpetra::CrsMatrix<SC,LO,GO,NO> >& A,
  Teuchos::ParameterList& params,
  const Teuchos::RCP<Tpetra::MultiVector<SC,LO,GO,NO> >& coords,
  const Teuchos::RCP<Tpetra::MultiVector<SC, LO, GO, NO> >& nullspace
);
```

. Tpetra objects take four template parameters, a scalar type (SC), a local ordinal type (LO), a global ordinal type (GO), and a node type (NO). Further, an RCP is a reference-counting, automatically deallocating pointer comparable to `std::shared_ptr`.

The constructor for the AmgX Adapter will then be called if Trilinos is installed with AmgX as a third party library. Trilinos can be configured with AmgX as a TPL by adding the following lines to a cmake configure script:

```
-D Trilinos_ENABLE_AmgX:BOOL=ON
-D AmgX_LIBRARY_DIRS:PATH="<path_to_amgx>/lib"
-D AmgX_INCLUDE_DIRS:PATH="<path_to_amgx>/include"
-D MueLu_ENABLE_Experimental:BOOL=ON
-D MueLu_ENABLE_AmgX:BOOL=ON
```

A user can pass in the AmgX configuration settings in two ways, either by passing in the name of a JSON file or by creating a sublist with AmgX configuration parameters. Either way, a sublist called "amgx:params" is created in the parameter list that is passed into `CreateTpetraPreconditioner`, and either "json file" is set or specific AmgX parameters are added to the sublist and set. The user must also specify "use external multigrid package" to be "amgx" in this parameter list. If this is set in the parameter list, `CreateTpetraPreconditioner` calls the `AMGXOperator` constructor:

```
AMGXOperator(
  const Teuchos::RCP<Tpetra::CrsMatrix<SC,LO,GO,NO> > &A,
  Teuchos::ParameterList &params
);
```

This constructor initializes the AmgX library and creates the configuration, resources, solver, matrix, and vector objects. Either the JSON file will be used to create the configuration object, or the "amgx:params" sublist will be converted into a single string to be used to create the object. In serial, three arrays are extracted from the Tpetra Matrix: column indices, row pointers, and data. These arrays are then passed into `AMGX_matrix_upload_all` along with the number of rows, block size, and number of nonzeros. The vectors are created using the resources object but the actual data for the vectors is not uploaded until the call to the apply method. A user can then solve a linear system by passing in right hand side and solution as `Tpetra::MultiVector` to the apply method. The vector data will

be uploaded to the AmgX vector objects and the linear system will then be solved using AmgX.

A unit test was added to the MueLu tests to ensure the AmgX adapter was working correctly. This tested AmgX on a two dimensional Laplacian matrix. The test checked that the AmgX matrix that was created was the correct size and that AmgX converged before the maximum number of iterations was reached. A JSON file with configuration settings was created and passed in to the unit test. The settings specified a V-cycle, preconditioned CG as the outer solver, block Jacobi as a smoother, and the direct coarse solver.

Further, in order to be able to use AmgX as a preconditioner with a Belos outer solver, the `BelosMueLuAdapter` was modified. The `BelosMueLuAdapter` takes a MueLu hierarchy object and turns it into a Belos object. This object can then be used as a preconditioner with a Belos solver object. The `BelosMueLuAdapter` was modified so that `AMGXOperator` objects could also be converted into Belos objects and used as preconditioners. A constructor that accepts an `AMGXOperator` object as a parameter was added. Further, the apply method was modified to check whether the `BelosMueLuAdapter` was initialized with a MueLu hierarchy or `AMGXOperator` and to then call the appropriate iterate or apply method.

4. Experimental Setup. Both two- and three-dimensional Laplacian matrices of various sizes were used to test the AmgX adapter. A right hand side equal to identity was chosen in each case, and the initial guess for the solution vector was randomized using a seed to ensure reproducibility. A performance comparison between standalone MueLu and MueLu using the AmgX adapter was conducted. Increasing matrix sizes were tested for this comparison. AmgX was set to run size 4 aggregation for 2D examples and size 8 aggregation for 3D. MueLu was set to run 2×2 brick aggregation for 2D and $2 \times 2 \times 2$ brick aggregation for 3D. Two configuration settings were used for the experiments. One setting used a single level multigrid hierarchy with Jacobi as a coarse solver. The other was set to use a Jacobi smoother with an exact solver on the coarse level. In this case, we wanted the coarsest level to have around 1000 rows. Because MueLu and AmgX do not have a comparable parameter to set this, *min_coarse_rows* was set to 500 in AmgX and *coarse: max size* was set to 1000 in MueLu. By setting the parameters in this way, MueLu and AmgX tended to produce the same number of multigrid levels on each problem size. The multigrid was set to run a V-cycle. All other parameters were set to the AmgX defaults. For the comparison between standalone MueLu and the adapter, a MueLu driver was given an input deck comparable to the settings passed in to AmgX. Belos was used in order to run CG as the outer iterative solver in both cases.

Remark. In our experiments,it took about 10 seconds for the AmgX library to initialize all its resources. This initialization could have been done at the program initialization, and thus we did not include this time in our timings comparisons between AmgX and MueLu.

Tests were run on the Shannon testbed. Shannon has Intel E5-2670 CPUs and various GPU devices. Tests were run on the "stella" queue of Shannon which has two K80 NVIDIA cards. CUDA version 6.5.14 was used with AmgX version 1.2.0 built on December 22, 2014.

5. Results.

5.1. A Comparison of MueLu and AmgX. The following are results from the comparison study briefly outlined in Section 4. In order to compare performance, the same parameters were used between the AmgX adapter and standalone MueLu. The Belos implementation of CG was used for the outer iterative linear solver in both cases. In all tables, time is measured in seconds. The setup time is either the time for the MueLu hierarchy to set up or the time for AmgX to set up the solver object once all data has been

# Rows	AmgX				MueLu			
	Setup	Solve	Total	Iters	Setup	Solve	Total	Iters
10000	0.0002	0.121	0.122	137	0.005	0.054	0.059	137

Table 5.1: Time results and number of iterations for comparison solving a 9-point Laplacian with 10000 rows. Both AmgX and MueLu were restricted to one multigrid level and a polynomial smoother was set to perform the coarse solve with the tolerance set to 10^{-6}.

# Rows	AmgX				MueLu			
	Setup	Solve	Total	Iters	Setup	Solve	Total	Iters
10000	0.026	0.048	0.071	19	0.052	0.024	0.076	14
40000	0.087	0.122	0.204	21	0.206	0.103	0.309	21
160000	0.096	0.388	0.483	32	0.745	0.534	1.379	31
640000	0.118	1.811	2.010	49	3.214	2.828	6.042	45
2560000	0.263	9.900	10.300	78	16.100	16.710	32.810	65

Table 5.2: Time and iteration results for comparison solving a 9-point Laplacian using polynomial smoother and direct coarse solve with tolerance set to 10^{-6}.

uploaded to the device. Solve time is the time that Belos took to converge and total is the setup time added to the solve time.

Table 5.1 shows the results from restricting both MueLu and AmgX to one multigrid level. Further, Jacobi was set as the coarse solver with a relaxation factor of 0.9 and Belos CG as the outer iterative solver. The tolerance was set to 10^{-6} and the matrix was a 9-point Laplacian with 10000 rows. 2×2 brick aggregation was chosen for MueLu and block size 4 aggregation was specified for AmgX. Because AmgX and MueLu produced the same number of iterations with these settings, it can be confirmed that AmgX is behaving as expected.

Table 5.2 shows the results from a scaling study done on 9-point Laplacian matrices. In this case, Jacobi was chosen as a smoother with a relaxation factor of 0.9 and a direct coarse solver was specified in both MueLu and AmgX. As mentioned previously, the multigrid was set so that the coarsest level would have around 1000 rows, and a V cycle was chosen. Again, Belos CG was chosen as the outer iterative method, and the tolerance was set to 10^{-6}. Also, block size 4 aggregation was chosen for AmgX and 2×2 brick aggregation was set for MueLu. The table include setup, solve, and total times for MueLu and AmgX as well as the number of iterations taken to converge in each case. It can be seen that despite taking more iterations to converge, AmgX did converge faster than MueLu on the three largest problem sizes. It can also be noted that the number of iterations remained fairly similar on all problem sizes for AmgX and MueLu. Further, the total time for AmgX was less than the total time for MueLu on all problem sizes.

Figure 5.1 shows the number of iterations and solution times for MueLu and AmgX as listed in Table 5.2. One can see that the total time for MueLu to complete increases faster than AmgX as the problem sizes grew. One can also see that the number of iterations were between AmgX and MueLu is pretty close. However, AmgX did tend to take more iterations than MueLu to converge.

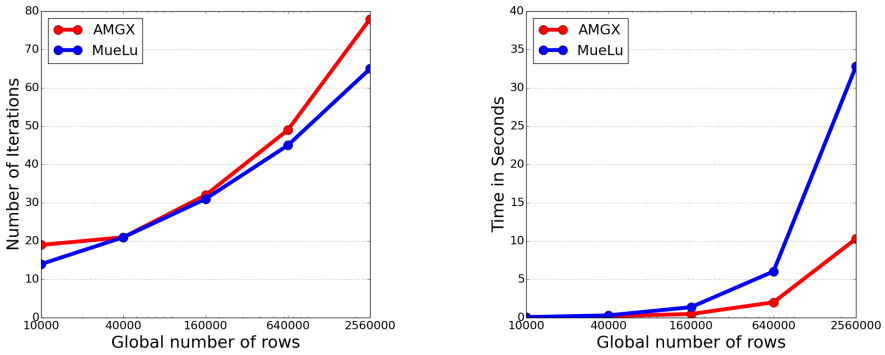


Fig. 5.1: Number of iterations (left) and total time (right) for 9-point 2D Laplacian matrices with Jacobi as smoother with direct coarse solver.

# Rows	AmgX				MueLu			
	Setup	Solve	Total	Iters	Setup	Solve	Total	Iters
10000	0.024	0.048	0.072	22	0.041	0.017	0.067	14
40000	0.090	0.128	0.218	24	0.160	0.074	0.234	22
160000	0.085	0.383	0.467	37	0.723	0.414	1.140	32
640000	0.110	1.690	1.800	56	3.760	2.360	6.120	47
2560000	0.194	9.050	9.240	87	16.100	13.800	29.900	68

Table 5.3: Time and iteration results for comparison solving a 2D Laplacian using Jacobi smoother and direct coarse solve with tolerance set to 10^{-6}.

Table 5.3 shows the results from running a scaling study on 2D Laplacian matrices. Here it can be seen that in most cases the setup and solve times for AmgX were less than those for MueLu. Aside from the different matrix, the configuration settings for this scaling study were the same as those described for Table 5.2.

Figure 5.2 shows the number of iterations and the total times for the scaling study run on 2D Laplacian matrices (Table 5.3). The plot shows that the total times for MueLu increased much faster than the total times for AmgX. Again, a similar number of iterations was seen between MueLu and AmgX on most problem sizes.

Table 5.4 displays the results from running a scaling study on 3D Laplacian matrices. Again, the configuration settings were the same as those in Table 5.2. However, instead of size 4 aggregation, size 8 was used for AmgX, and $2 \times 2 \times 2$ brick aggregation was used for MueLu. In this case, the solve times remained fairly similar between AmgX and MueLu. However, as problem sizes increased, MueLu setup times increased more rapidly than those for AmgX.

Figure 5.3 shows the number of iterations and the total times from Table 5.4. Very similar trends are seen here as were noted in Figures 5.1 and 5.2. Here, AmgX tended to need quite a few more iterations than MueLu to converge.

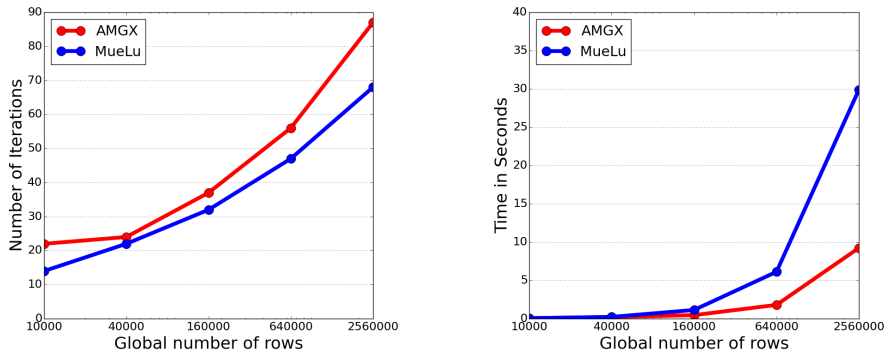


Fig. 5.2: Number of iterations (left) and total time (right) to setup and solve a 2D Laplacian using Jacobi smoothing and direct coarse solver.

# Rows	AmgX				MueLu			
	Setup	Solve	Total	Iters	Setup	Solve	Total	Iters
8000	0.041	0.041	0.235	16	0.030	0.015	0.045	10
64000	0.046	0.117	0.312	22	0.234	0.073	0.308	14
512000	0.087	0.895	1.090	34	2.220	0.860	3.080	21
4096000	0.302	9.120	9.310	51	26.500	9.530	36.000	31

Table 5.4: Time and iteration results for comparison solving a 3D Laplacian using Jacobi smoother and direct coarse solve with tolerance set to 10^{-6}.

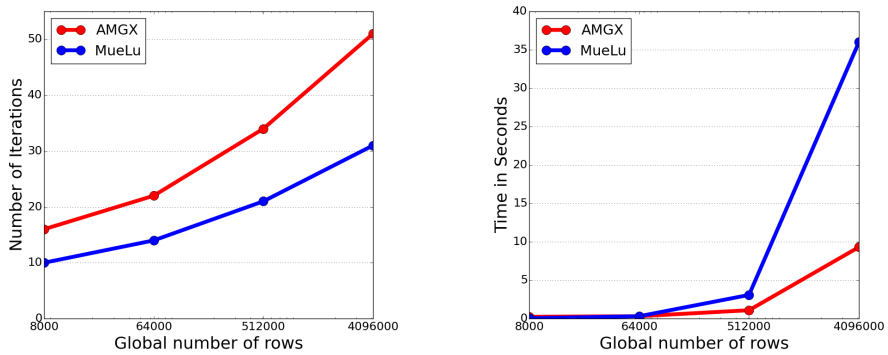


Fig. 5.3: Number of iterations (left) and total time to setup and solve (right) to converge for a 3D Laplacian using Jacobi smoothing and direct coarse solver.

6. Conclusions. In this work we created an adapter to AmgX inside of MueLu. The adapter allowed for serial use with AmgX and allowed AmgX to be used as a preconditioner and solver or as a preconditioner with Belos for the outer iterative method. From a scaling study it was shown that AmgX and MueLu as preconditioners resulted in a comparable number of iterations to converge with Belos as the outer iterative method. Further, it was seen that on larger problem sizes, AmgX resulted in faster setup and solve times, and demonstrated better scalability.

7. Future Work. An immediate area of future work would be to modify the adapter to allow for MPI parallel usage. While the adapter currently can be run in serial utilizing one device or GPU, AmgX requires quite a bit more information to allow for MPI parallelism. We were able to extract the necessary information from the Tpetra objects, but AmgX requires a packed local matrix for the call to `AMGX_matrix_upload_all`. However, AmgX requires that the rows belonging to each process be reordered. For instance, on a certain process, AmgX requires that column indices be reordered in such a way that global diagonal elements come first. Following the diagonal elements, the remaining column indices must be ordered based on which neighboring process they belong to. After this reordering, the rows must be ordered in such a way that rows with no connections to neighbors come first. In addition to these reordering steps, a new mapping from local to global is needed to keep track of how the rows are now ordered. Because the rows of Tpetra matrices can be in arbitrary order, the adapter would need to convert the matrix into this packed format required by AmgX and ultimately store it twice creating a large amount of overhead in order to successfully utilize MPI parallelism with AmgX.

Next steps for the AmgX adapter include making the adapter compatible with Epetra objects. Currently there are methods to create MueLu preconditioners for both Epetra and Tpetra objects, and ideally, in the future, a user would be able to call either one of these functions with AmgX configuration parameters and invoke the AmgX adapter.

Further, we would like to create a FENL (Finite Element Nonlinear Solver) example using the AmgX adapter. Essentially, the goal would be to accomplish assembly, preconditioner setup, and solve entirely on the GPU. The examples and tests in this paper involved problem assembly on the CPU followed by a call to upload the data to the GPU before preconditioning on the device and solving on the CPU.

A. AmgX Configuration Files. The following is a MueLu input deck which would create an AmgX adapter object.

```
<ParameterList name="MueLu">
  <Parameter name="verbosity"            type="string" value="high"/>
  <Parameter name="use external multigrid package" type="string" value="
      amgx"/>
    <ParameterList name="amgx:params">
    <Parameter name="json file"          type="string" value="laplace2d.
        json"/>
  </ParameterList>
</ParameterList>
```

The below listing is an AmgX configuration file. It sets up a Jacobi smoother with a direct coarse solver and size 4 aggregation.

```
{
   "config_version": 2,
   "solver": {
      "solver"             : "AMG",
```

```
      "cycle"              : "V",
      "algorithm"          : "AGGREGATION",
      "selector"           : "SIZE_4",
      "max_iters"          : 1,
      "min_coarse_rows"    : 500,
      "smoother" : {
         "solver"          : "BLOCK_JACOBI",
         "relaxation_factor" : 0.9,
         "monitor_residual" : 1,
         "scope"           : "jacobi"
      },
      "presweeps"          : 1,
      "postsweeps"         : 1,
      "print_grid_stats"   : 1,
      "monitor_residual"   : 1,
      "scope"              : "amg",
      "print_config"       : 1,
   }
}
```

B. MueLu **Input Deck.** The following is an input deck used to run MueLu examples. The XML file contains configuration settings for 2×2 brick aggregation, a Chebyshev smoother, and a direct solver.

```
<ParameterList name="MueLu">
  <Parameter name="verbosity"        type="string" jvalue="high"/>
  <Parameter name="number of equations" type="int" value="1"/>
  <Parameter name="max levels"       type="int"  value="10"/>
  <Parameter name="coarse: max size" type="int"  value="1000"/>
  <Parameter name="multigrid algorithm" type="string" value="unsmoothed"/>
  <Parameter name="aggregation: type" type="string" value="brick"/>
  <Parameter name="aggregation: brick x size" type="int" value="2"/>
  <Parameter name="aggregation: brick y size" type="int" value="2"/>
  <Parameter name="aggregation: brick z size" type="int" value="1"/>
  <Parameter name="smoother: type"   type="string" value="RELAXATION"/>
  <ParameterList name="smoother: params">
    <Parameter name="relaxation: type" type="string" value="Jacobi"/>
    <Parameter name="relaxation: damping factor" type="double" value="0.9"
        />
    <Parameter name="relaxation: sweeps" type="int" value="1"/>
  </ParameterList>
</ParameterList>
```

REFERENCES

[1] *Trilinos project web site.* http://trilinos.org, 2015.

[2] C. G. Baker and M. A. Heroux, *Tpetra, and the use of generic programming in scientific computing*, Scientific Programming, 20 (2012), pp. 115–128.

[3] E. Bavier, M. Hoemmen, S. Rajamanickam, and H. Thornquist, *Amesos2 and Belos: Direct and iterative solvers for large sparse linear systems*, Scientific Programming, 20 (2012), pp. 241–255.

[4] W. L. Briggs, S. F. McCormick, and V. E. Henson, *A multigrid tutorial*, SIAM, 2nd ed., 2000.

[5] J. Gaidamour, J. Hu, C. Siefert, and R. Tuminaro, *Design considerations for a flexible multigrid preconditioning library*, Scientific Programming, 20 (2012), pp. 223–239.

[6] M. Heroux, R. Bartlett, V. Howle, R. Hoekstra, J. Hu, T. Kolda, R. Lehoucq, K. Long, R. Pawlowski, E. Phipps, A. Salinger, H. Thornquist, R. Tuminaro, J. Willenbring, A. Williams, and K. Stanley, *An overview of the Trilinos package*, ACM Trans. Math. Softw., 31 (2005).

[7] M. A. Heroux and J. M. Willenbring, *A new overview of the Trilinos project*, Scientific Programming, 20 (2012), pp. 83–88.

[8] M. F. Hoemmen, J. J. Hu, and C. S. Siefert, *Ifpack2: incomplete factorizations, relaxations, and domain decomposition library.* `http://trilinos.org/packages/ifpack2`, 2014.

[9] M. F. Hoemmen, C. Trott, and M. A. Heroux, *Tpetra: Next-generation distributed linear algebra.* `http://trilinos.org/packages/tpetra`, 2014.

[10] NVIDIA, *AmgX Reference Manual: API Version 2*, 2014.

[11] L. Olson, J. Schroder, and R. Tuminaro, *A general interpolation stategy for algebraic multigrid using energy minimization*, SIAM Journal on Scientific Computing, 33 (2011), pp. 966 – 991.

[12] A. Prokopenko, J. J. Hu, T. A. Wiesner, C. M. Siefert, and R. S. Tuminaro, *MueLu users guide 1.0*, Tech. Rep. SAND2014-18874, Sandia National Labs, 2014.

[13] M. Sala and R. S. Tuminaro, *A new Petrov-Galerkin smoothed aggregation preconditioner for non-symmetric linear systems*, SIAM Journal on Scientific Computing, 31 (2008), pp. 143–166.

[14] P. Vaněk, J. Mandel, and M. Brezina, *Algebraic multigrid based on smoothed aggregation for second and fourth order problems*, Computing, 56 (1996), pp. 179 – 196.

A TESTING FRAMEWORK FOR A HYBRID TRIANGULAR SOLVER

WILLIAM B. HELD* AND ANDREW M. BRADLEY†

Abstract. When developing new software solutions, rigorous performance and correctness testing helps to produce a quality production code. This paper outlines the functionality of a framework for testing the new research multithreaded sparse triangular solver HTS against a large set of matrices.

1. Introduction. The goal for this project was to create a testing framework which would allow for performance and correctness testing of HTS on full solves on matrices from the University of Florida's test set [2].

HTS is a prototype threaded triangular solver. It has three phases: an expensive symbolic analysis, a fast numerical phase, and a fast solve phase. The symbolic analysis is run once for a fixed graph. HTS is effective when a problem requires solving in sequence many triangular systems having the same matrix T or a sequence of matrices T_i sharing a common nonzero pattern. HTS decomposes T into three parts: a level-scheduled (LS) triangle, a large matrix-vector product (MVP) block that scatters the (LS) solution, and a data-parallel (DP) triangle. This decomposition tends to make the DP triangle substantially denser than the original matrix. Threaded algorithms are implemented for each of these three blocks. HTS exposes parameters, but its default parameter values were used in this study.

Performance tests are important for two reasons when looking at the applications of a triangular solver. The variety of the University of Florida test matrices provides a large sampling of problems to help determine what types of problems show good performance with the algorithm and which problems contain interesting issues which cause the software to struggle, or on which the solver is incorrect. This data can lead to bugfixes or simply give insight into what the software is good at. Secondly, timing full solves using HTS provides a good test of how well the software reduces the bottleneck of a triangular solve when included in a larger mathematical operation.

To achieve this goal the framework has five fundamental steps. Initially, the University of Florida matrices were limited to matrices that would be possible and useful to test upon. In order for a triangular solver to be used to perform full solves the matrix A is put through LUPQR factorization. Then, the full solves are performed and timed using HTS as the operator. HTS is compared to Intel's Math Kernel Library (MKL)[1] on the same problems to give the user a point of reference for performance numbers. Finally, the output from the timings is parsed and used to create meaningful figures which could be used in a research paper or presentation.

2. Framework Structure.

Creating Candidate list. While the goal was to test HTS against the University of Florida matrices, many of these matrices were not used in testing for two fundamental reasons. They were either not big enough to be worthwhile for testing or they were numerically singular. To trim the list down to only those which were suitable for testing, the entire University of Florida test matrices were passed through a bash script of Matlab functions which first eliminates insignificant matrices and then eliminates matrices which cannot be prepped for triangular solves.

*Albuquerque Academy; New York University Abu Dhabi, wbh230@nyu.edu
†Sandia National Laboratories, ambradl@sandia.gov

Using the UFget library in Matlab, the index provided by the University of Florida for the matrices is accessed. This data is used to trim off those matrices that are too small (≤ 1000 rows) to have any meaningful improvements through threading and to check whether the matrices are structurally full rank. This eliminates the meaningless matrices and the impossible matrices which can be eliminated purely from the index values. Then, that list is taken and begins factorizing each matrix into factors of L (Lower Triangular), U (Upper Triangular), P (Row Permutation Matrix), Q (Column Permutation Matrix), and R (Diagonal Scaling Matrix) from the UMFPACK `lu` function within Matlab while under limits for RAM and runtime. Those UMFPACK factors are such that $L*U = P*(R\backslash A)*Q$. This can be substituted for A in the equation $A * \vec{x} = \vec{b}$. This allows the equation to be solved as two triangular solves for the forward and backward substitution steps.

The RAM limit eliminates those matrices which have too large of a fill factor to be stored on the machine and the runtime eliminates those problems for which LU factorization is too great of a workload. These limits are placed on Matlab from the Bash wrapper using the Unix command `ulimit` which provides easy access to resource management for a particular Bash window. It also allows for easy modifications of the memory and time limits for the script for different machines. If the while loop for the factorization is performed internally in Matlab a time limit cannot be set as the entire Matlab process will be considered one bash process. Therefore, each factorization is opened as an individual Matlab instance as shown in 2.1.

```
ulimit -S -v 4000000
ulimit -S -m 4000000
ulimit -S -t 20
i=1
while 1; do
  matlab -nodisplay -nodesktop -r
    "addpath('/home/wbheld/Documents/MATLAB/UFget', '/home/wbheld/wbheldCompton');
     C = load('FinalCandidates.txt'); load('Second_Round.m', '-mat');
     try gts_setupMOD('correctness_prep', C($i)); F($i) = C($i);
     end; save('Second_Round.m', 'F'); quit"
  i=$(($i+1))
done
```

Fig. 2.1: Bash Wrapper

The candidates that are successfully factorized are converted into CRS format to be used as inputs in C++. Additionally, a dense set of vectors of the same size as the original matrix is generated randomly to be used as solution vectors in the equation $\mathbf{A}\vec{x} = \vec{b}$. One of these vectors will be used in performance testing and either one or three of the vectors will be used in correctness testing. A baseline for error is established using the Matlab backslash function ($\vec{x}_{\text{true}} = \mathbf{A}\backslash\vec{b}$). Each of these values is stored into a data file which will be input into the C++ performance testing.

Performance testing. The performance tests themselves are the core of the framework. The framework outputs time per solve, speedup as compared to the baseline, time for symbolic analysis, time for numerical processing, and the relative error of the solve in a chart along with key statistics about the matrix itself and how the problem was compiled in this instance. The goal with this output is to provide all data needed for performance statistics or troubleshooting errors originating from compilation or matrix conditions.

First, the data file is read with the L and U factors being stored as CRS (Compressed Row Storage) matrices while $\vec{b}$ and $\vec{x}_{\text{true}}$ are stored as C++ standard vectors. The data for compiling options, as well as the University of Florida index number, is recorded in order to preserve the options which created the outputs. The number of non-zeros within the Lower

and Upper triangular factors are recorded as well, since those cannot be retrieved from the University of Florida index based off of the index number.

A baseline for performance is set up based off of one of two basic triangular solvers, either a basic serial solve or Intel's Math Kernel Library sparse triangular solve. A serial full solve is performed to create a baseline for speedup.[1]. For a more accurate comparison of speedup in a scientific computing environment, the problem is run through Intel's Math Kernel library as a baseline of expected performance for a threaded commercial triangular solver.[2] These baseline solves have their relative error and times per solve recorded. If MKL is used than the speed of MKL serves as the factor of 1 for speedup, otherwise the hand-coded serial solve will be used to ensure that the testing framework performs even upon a basic consumer computer.

After the baseline has been set, whether it is the serial solve or MKL, the problem is run with HTS. Both the Upper and Lower triangles go through symbolic analysis and that symbolic analysis is timed. Then, the output goes through the numerical phase which is kept as a separate timing. It is important to note that the symbolic analysis and numerical phase are not taken into account in the calculations of speedup. This means that one must always consider how many solves will be performed for the speedup to be worth the cost of the symbolic and numerical analysis.

The last timing is that of the actual algorithm. HTS is given the outputs from the analysis phase along with the original permutation and scaling vectors. HTS's solve is then compared to the baseline solve to generate the speedup value. This process is repeated for a subset of 1 to 32 threads on a 16 core Sandy Bridge CPU, 1 to 40 threads on a 20 core Ivy Bridge CPU, and 1 to 228 threads on an Intel Xeon Phi MIC. The performance may not see significant improvement once hyperthreading on the CPU is activated; however, the performance is checked in order to confirm that hyperthreading does not degrade performance substantially. This series of tests records the thread count, time per solve, speedup, and relative error.

```
> 236
 USE_MKL USE_SERIAL_SPARSE USE_DENSE_ROWS SORT_BLOCKS USE_P2P
 min_block_size 64 serial_block_size 64 max_level_set_bottlenecks 1073741824 min_dense_density 0.75
ls_blk_sz 1 min_lset_size 10 min_lset_size_scale_with_nthreads 0 profile 0
nrhs 1 nnzL 3322821 nnzU 3390972
n 3140 |lsis| 450 |dpis| 2690
n 3140 |lsis| 544 |dpis| 2596
thr sec/solve speedup prepr/s| repr/s||   relerr solver
 -1 1.380e-02    0.67     0.00     0.00 3.62e-12    sfs
  0 9.303e-03    1.00     0.00     0.00 3.52e-12    mkl
  1 5.581e-03    1.67    24.13     9.81 3.87e-12    gts
  2 2.918e-03    3.19    18.91    10.31 3.41e-12    gts
  4 1.941e-03    4.79    16.87     8.54 3.37e-12    gts
  8 1.659e-03    5.61    16.56     6.22 3.33e-12    gts
 12 1.708e-03    5.45    16.62     5.86 3.55e-12    gts
 16 1.723e-03    5.40    16.84     5.45 3.56e-12    gts
 24 1.896e-03    4.91    17.48     5.00 3.61e-12    gts
 32 1.943e-03    4.79    22.37     5.70 3.44e-12    gts
<
```

Fig. 2.2: Sample Output

With all of the appropriate data recorded, the testing framework outputs the data in the format shown in 2.2. In the output, the times for symbolic and numerical analysis

[1]If the GCC compile instructions at the top of the file are used, then a hand-coded serial solver will be the baseline for speedup throughout.

[2]For more final testing and use of MKL, Intel compilers should be used as opposed to GCC.

are transformed to quantities which are more convenient for analysis. Symbolic analysis is compared to the time it takes to perform a serial solve while the numerical analysis is compared to the time it takes to perform a parallel solve.

Parsing. Performance tests are scripted in batches, with the outputs stored in a text document whose title tells the user the date the tests were run, the machine they were run on, and the range of University of Florida index values for that particular batch. While such a text document contains all the data needed for a thorough analysis, the data is of little use within that format and it is necessary to read through the data in order to have presentable results. The parser, which processes the data, was written within Matlab for ease of transition directly into graph creation once the data is stored.

The parser begins reading data when it comes across the symbol '>'. This allows the user to input their own notes into the record without breaking the parser. Each variable is stored within a structure which contains data for a certain run. This structure can then be used as the access point for an entire performance testing batch rather than accessing each variable separately.

The parser stores the index number which will be used to draw additional data from the index. The name of the matrix is drawn from the UFget index so that performance can more easily be associated with the application matter of a particular problem and in case the index numbers change in future iterations of the University of Florida set. All compiling options are stored in case they are needed to isolate a bug. The number of non-zeros from the original matrix is pulled from the index. The size of the matrix is drawn from the mark '`N`'. The number of non-zeros for both the Lower and Upper triangular factors are pulled from the marks '`nnzL`' and '`nnzU`'. These 3 records will be the key graphical comparisons for performance as they are the most numerically significant comparisons for speedup. Every section from the chart is then recorded as an individual variable in the structure. The parser knows to move to the next matrix when it comes across the symbol '<'.

Graph Creation. While the parsed data is usable for calculations in the Matlab format, it would be meaningless in a presentation without copious amounts of explanation. Therefore, graphing is necessary to make the data easily memorable and meaningful to the viewer whether in a written report or a presentation. For this framework, speedup compared to number of non-zeros and total size of the matrix were graphed in order to analyze how dependent on size and sparsity the algorithm is. Even with a quick glance this gives the reader an idea of how well HTS performs in the most general terms, with deeper analysis still available stored in the Matlab structure.

Examples of analysis of the subset of the University of Florida matrix collection we have assembled are shown in 2.3 and 2.4. 2.3 shows the speedup relative to MKL's serial triangular solver on a 20-core, 2-hyperthread CPU for 10 and 40 threads (y-axis) as it relates to the size of the matrix and number of non-zeros in the matrix factors (x-axis). For very small matrices and some large ones, speedup is below 1. 2.4 shows speedup on the Intel Knights Corner MIC compared to the same statistics. On this architectur and with the OpenMP settings used, 56 threads corresponds to using one quarter of the cores, and 228 threads corresponds to using all cores. These comparisons allow for quick analysis of an extremely large data set over two architectures and allows for much more general trends to be checked. It is important to note that the purpose of this report is to describe and evaluate a test framework, not yet to evaluate HTS. For that reason, the set of tests shown in 2.3 and 2.4 is dominant in problems that are smaller than those of primary interest to HTS, and has only a subset of the UF collection of particular interest.

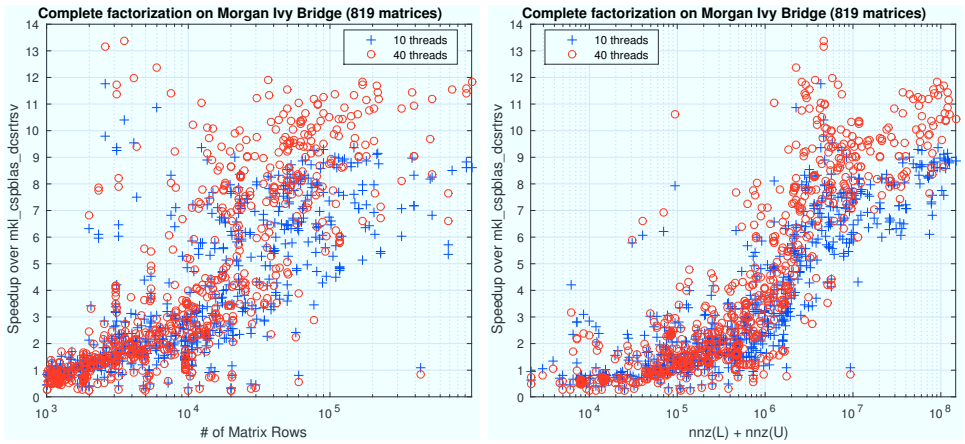


Fig. 2.3: Complete Factorization of UF matrices on Ivy Bridge

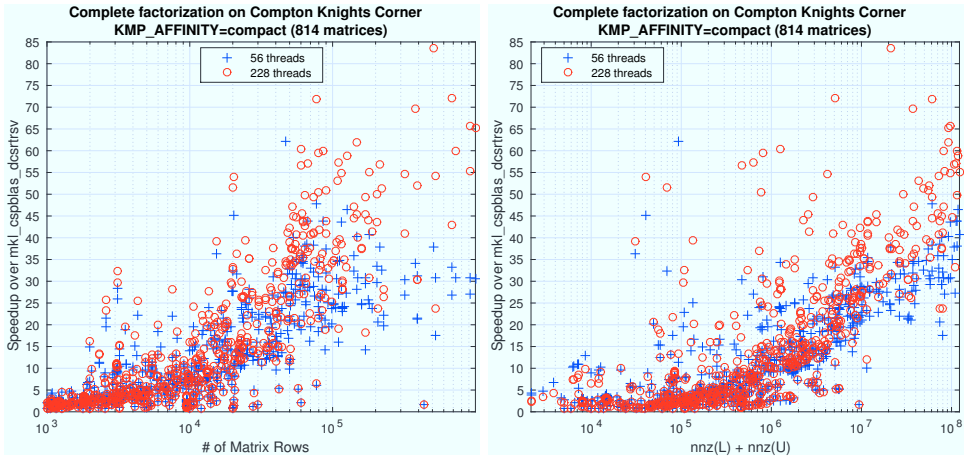


Fig. 2.4: Complete Factorization of UF matrices on Knights Corner

3. Conclusion. The goal of this project was to create a framework which when given University of Florida's matrices could time, record, and analyze the performance of HTS in complete solves against as many worthwhile candidates as possible. The framework successfully creates a useful test set from University of Florida's larger set of matrices. It performs full solves while timing and recording all valuable aspects of the solve. Finally, It breaks down the raw outputs of the timings and is capable of turning raw outputs into meaningful figures which could be used in a report while also providing enough of the source data to create more specified figures if the need arose.

REFERENCES

[1] *Intel Math Kernel Library. Reference Manual*, Intel Corporation, 2009.
[2] T. A. Davis, *The University of Florida sparse matrix collection*, NA DIGEST, (1997).

VISUALIZATION FOR MULTIGRID AGGREGATION

BRIAN M. KELLEY*, CHRIS M. SIEFERT†, AND RAY S. TUMINARO‡

Abstract. The MueLu package in Trilinos is a modern, parallel implementation of the algebraic multigrid technique. A key part of multigrid is a process called aggregation, where nodes in a mesh are organized into small groups. Aggregation routines are sophisticated and are subject to many constraints. An easy to use, configurable tool to visualize the aggregation process using VTK is therefore useful and is described in this article.

1. Introduction. MueLu is the current algebraic multigrid library in the Trilinos family of libraries. It speeds up solving linear algebra problems of the form $A\vec{x} = \vec{b}$, where A is sparse and $\vec{b}$ is one or more column vectors. Large problems in this form are solved as part of the finite element method (FEM). In a FEM model, space is partitioned into a mesh. For a single partial differential equation problem with N nodes making up the mesh, the matrix A is NxN; each row and each column correspond to a node. For each connection (edge) in the mesh, a nonzero entry is placed in A where the row is one endpoint of the edge and the column is the other endpoint. The entry's value is an approximate evaluation of some differential expression at a fixed point in time. The vector $\vec{b}$ contains the boundary conditions and other force terms for the problem's domain. After solving the problem, $\vec{x}$ contains the scalar values for all the nodes in the mesh that satisfy the original differential equation. Implicit finite element simulations need to solve at least one large linear system every time step, so a high performance sparse solver is very important for simulation.

MueLu accelerates the iterative process of solving these systems by repeatedly coarsening the mesh and using coarse approximations to improve the convergence rate of an iterative solver. MueLu keeps track of a set of levels that each contain a (progressively smaller) version of the matrix A and other data that is used in the multigrid process. Starting from Level 0 (the finest level with the largest matrix), MueLu creates small groups (aggregates) of nodes in the mesh. The center of each aggregate becomes a node in the next level's mesh. This means that the number of nodes and edges in the next level are smaller, so the problem becomes easier to solve. MueLu applies a few iterations of a smoother to the new coarse matrix, which generates an approximate solution for each level. More levels are created until the size of the matrix reaches a small enough size to solve very quickly with a direct solver. Then that exact coarse solution is used to eliminate most of the error in the next finest level's approximate solution. The solutions are smoothed again after being corrected. Once this process reaches Level 0, a good approximation of the solution to the original problem is available. This result can then be passed into an iterative solver like Belos, and converges to an exact solution very quickly, [2].

The aggregation algorithms in MueLu are complicated. Nodes near the edge of the mesh have to be handled carefully in order to make sure that aggregates end up with a nearly equal number of nodes. Some simulation problems have cracks and discontinuities in the mesh, so that certain node connections aren't allowed. Because of these challenges, it is useful to the MueLu developers to have an easy way of viewing the aggregation process on each level. The goal of this visualization project was to show how aggregates appear on the mesh. The code was to output widely compatible VTK files encoded in XML, and to support both serial and parallel problems.

*Texas A&M University, kelb150@tamu.edu
†Sandia National Laboratories, csiefer@sandia.gov
‡Sandia National Laboratories, rstumin@sandia.gov

2. Aggregates Visualization. The aggregates visualization project began with a discussion about the different ways that aggregates could be represented visually. The three methods, or "styles", that were prioritized were Point Cloud, Jacks and Convex Hulls. Each one is described below. The styles have various strengths and weaknesses, which means the user can choose one based on what kind of information they are looking for.

2.1. VTK Files. In all three of these modes, all geometry is defined in a .vtu (VTK unstructured grid) format. The data is in XML format. XML-encoded VTK is a more modern and powerful format than legacy ASCII VTK files, but is still compatible with all software that uses VTK. In this format, a list of points are defined with scalar fields and coordinates. For aggregates visualization, each point's unique node ID, aggregate ID and owning processor are stored in the VTK file. Then a list of cells are created. A cell can be any geometric primitive, like a point, line or triangle. Each cell is defined from a set of points. For example, a line segment is constructed out of two points. In ParaView, users can select one of the points' scalar quantities to map to a color gradient. Each cell is then colored based on the data of its points.

2.2. Improvements. Before this project, MueLu did have a simple aggregate visualization capability, but it was more limited. The "AggregationExportFactory" class that was modified during this project was able to output a raw table of aggregate IDs and the nodes that belong to each aggregate. To invoke the AggregationExportFactory, MueLu had to be configured with a parameter list that explictly created the factory object. Then a separate python script loaded the aggregate table as well as a coordinates array and used the python library qhull to generate convex hulls for each aggregate. Finally, the script took the convex hull data and wrote them to a legacy VTK file. The new aggregates visualization improves upon this functionality in several ways. It is supported by the "easy parameter list" interface, where an interpreter within MueLu automatically creates all needed factories. It is more configurable, allowing multiple geometric "styles" of displaying aggregates, instead of only the convex hulls. It can also output graph edges from the fine and coarse levels. Lastly, it gathers all needed information directly from the level (including node coordinates), so it runs completely during the hierarchy setup and requires no post-processing to produce a VTK file.

2.3. Point Cloud. "Point Cloud" is the simplest aggregates visualization style. Each node is represented by a single point, and no additional geometry is created to represent aggregates. Aggregates can be viewed, however, by coloring each point based on its aggregate ID. The benefits of this style are small file size, very fast geometry creation time and a less cluttered appearance than the other styles for large 3D meshes. The main downside of Point Cloud is that it is very difficult to tell which aggregate a point belongs to because of the smooth default color gradient used by ParaView. This can be easily remedied by loading a custom random colormap where consecutive aggregates have completely different colors. Figure 2.1 shows a 2D point cloud, and Figure 2.2 shows a 3D one. In both figures, points are colored by aggregate.

2.4. Jacks. "Jacks" is another simple visualization scheme. To produce this style, each aggregate is required to have exactly one "root node", preferably at the center of the aggregate. MueLu's default uncoupled aggregation code does this. In Jacks, line segments are drawn from the root node to all other nodes in the aggregate. Then points are added to the free ends of the line segments. The result looks similar to toy jacks. This style is the only one that explicitly shows the root node. It also complements the coarse graph overlay because the fine root node can be compared directly to the coarse node at the centroid of the aggregate. Figures 2.3 and 2.4 show the Jacks style in 2D and 3D, respectively. Figure

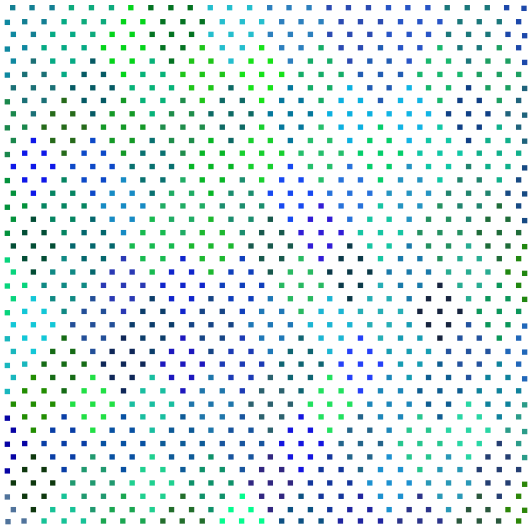


Fig. 2.1: 2D Point Cloud

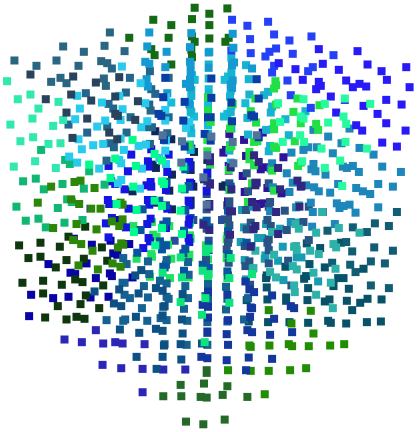


Fig. 2.2: 3D Point Cloud

2.5 shows Jacks with the coarse graph (red). Notice that the graph connections are at the centers of aggregates. The coarse graph overlay will be explained in Section 2.6.

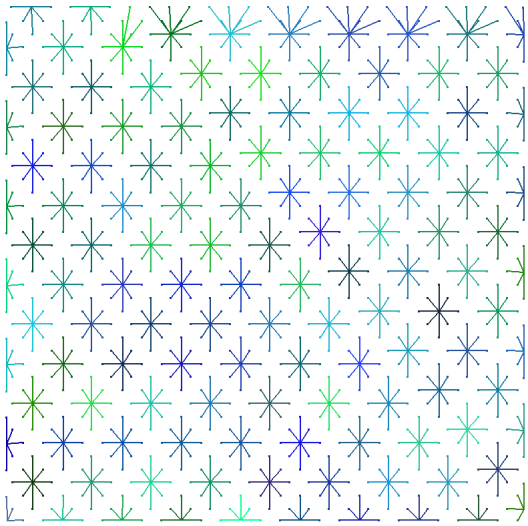


Fig. 2.3: 2D Jacks

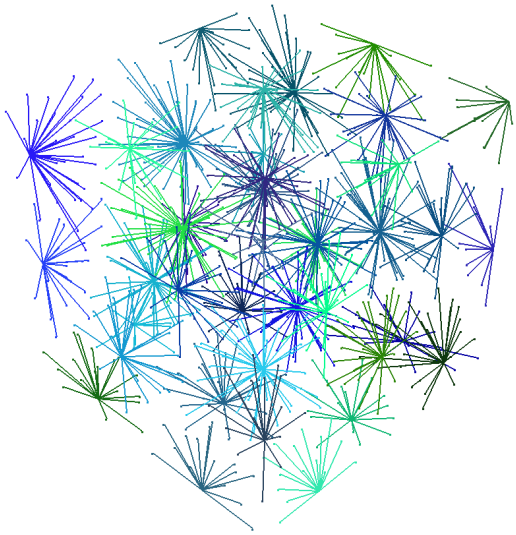


Fig. 2.4: 3D Jacks

2.5. Convex Hulls. A convex hull is the smallest convex polygon or polyhedron that includes or contains a given set of points. In this case, the points are all the nodes in each aggregate.

To construct 2D convex hulls, an implementation of the simple Jarvis March algorithm was created. This algorithm first finds the point(s) with the minimum x coordinate, and out of those the one with the minimum y coordinate. This point is known to be a vertex in the convex hull. Then a ray is drawn from that point to some other point in the aggregate. If any point is on the left side, the ray is rotated to pass through that point instead. This is

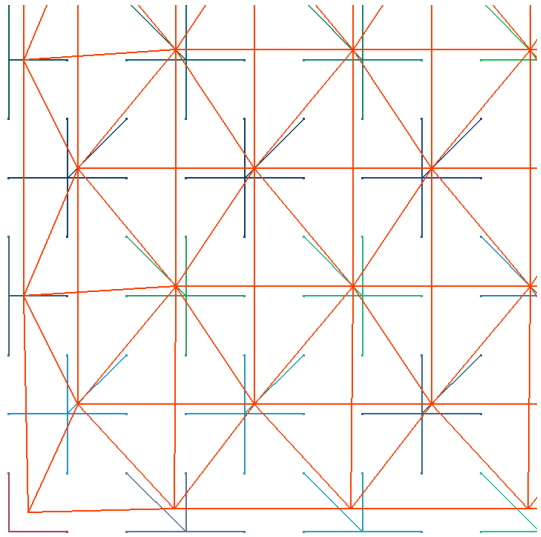


Fig. 2.5: Jacks with Coarse Mesh

repeated until no points are on the left side. The ray now passes through the next point in the convex hull. The scan is repeated for the new point to determine the third point. This procedure is repeated over and over, constructing the convex hull in a clockwise direction. Eventually the starting point is reached again, which signals that the convex hull is complete. On average, Jarvis March works in $O(n^2)$. Since the aggregates rarely contain more than 10 points, the algorithm runs very quickly in practice. Figure 2.6 shows 2D convex hulls.

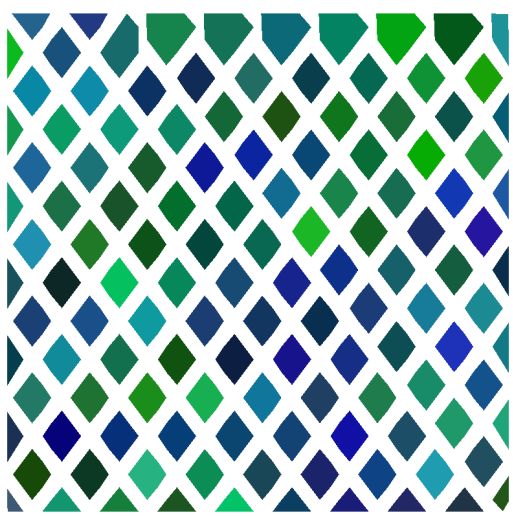


Fig. 2.6: 2D Convex Hulls

3D convex hulls are more difficult to generate efficiently. The Quickhull algorithm was

implemented as described in a blog post by Thomas Diewald [1]. Figure 2.7 shows 3D convex hulls.

Fig. 2.7: 3D Convex Hulls

2.6. Parallel VTK Files. A significant benefit of using ParaView and VTK files for visualizing aggregates is that VTK features a very simple way of reading and writing sets of data in parallel. Since most large problems in MueLu are run with more than one MPI rank, it is important that the visualization is able to show how different aggregation algorithms handle processor boundaries. When a problem is set up, nodes are distributed evenly across the threads, so that each processor "owns" all the nodes in a particular region of the mesh.

In order to visualize the nodes on all processors together, a parallel VTK file (.pvtu) is created. This file contains a list of other files from which to read point and cell data. This means that each MPI rank in the problem can write to its own VTK file, while only the root processor (processor #0) is responsible for writing the short pvtu file. This technique has performance benefits, since communication among processors is kept to a minimum and the visualization geometry can be calculated in parallel. Figure 2.8 shows the content of the vtu file from processor 1 in a 4-processor problem, and Figure 2.9 shows the combined visualization of all processors. In these images, points are colored by processor ID.

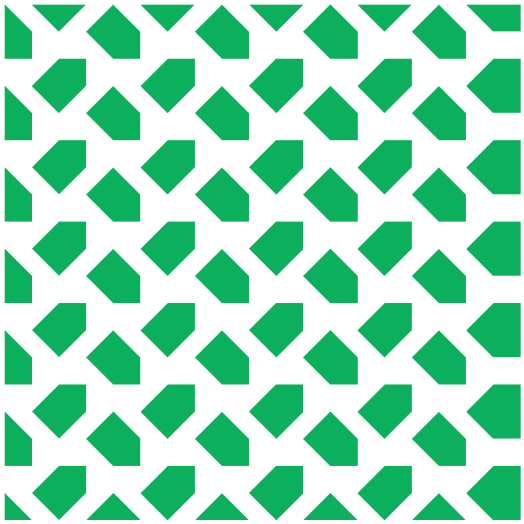


Fig. 2.8: Aggregates on Processor 1

Notice that the aggregates do not extend into the processor boundaries. Uncoupled aggregation only allows aggregates to exist within the boundaries of a single processor.

2.7. Graph Overlay. The visualization code also allows the user to request that edges in the graph be drawn along with the aggregates. Both the fine and coarse graphs can be displayed. The fine graph shows the connections among all the nodes that were grouped together into aggregates, while the coarse graph shows the result of coarsening the mesh through aggregation. Figures 2.10 and 2.11 are examples of the fine and coarse graph overlays, respectively. They are drawn on top of the 2D convex hulls. This problem is also running on 4 processors, and level 0 is shown.

Note that the fine mesh is a very simple square grid for this example problem. The aggregates do not cross processor boundaries, but the grid itself is unaffected by being split among processors. The coarse mesh makes the role of the aggregates clear. A node is placed at the center of each aggregate, and connections are created between nodes and other nearby nodes. The aggregation routine has done a good job, since the graph is mostly regular and

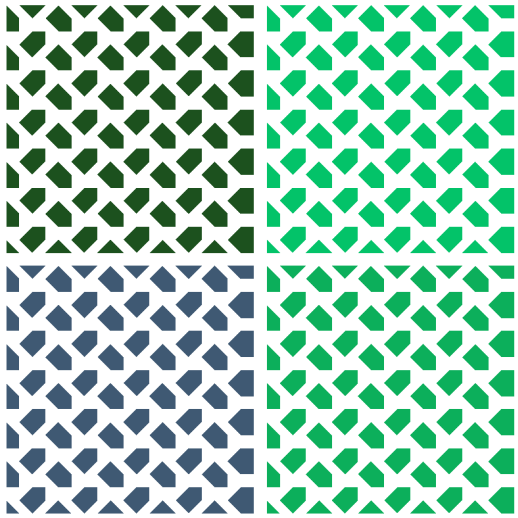


Fig. 2.9: Aggregates on 4 Processors

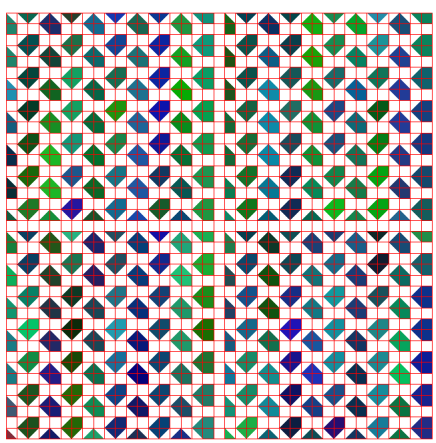


Fig. 2.10: Fine Graph Overlay

nodes are evenly spaced.

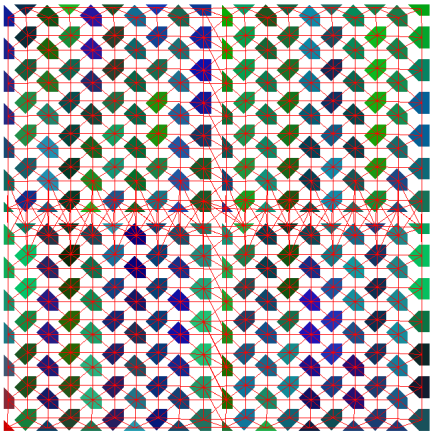


Fig. 2.11: Coarse Graph Overlay

2.8. Usage. To use this code, a few parameters must be set in the MueLu parameter list:

- aggregation: export visualization data
  Must be set to true to enable the AggregationExportFactory.
- aggregation: output filename
  The name of the destination VTK file.
- aggregation: output file: agg style
  The desired style of aggregates visualization, e.g. "Jacks"
- aggregation: output file: fine graph edges
  Whether to output the edges of the fine level graph.
- aggregation: output file: coarse graph edges
  Whether to output the edges of the coarse level graph.
- aggregation: output file: build colormap
  Whether to output a custom colormap in XML format for ParaView

These parameters can be set in the "global" parameter list, which will output VTK files for every level. Alternatively, they can be set in "level n" sublists, which would only produce the files for level n. If an XML factory list is being used, these parameters can be applied to the AggregationExportFactory for the same effect.

The given filename can contain some special tokens, where every occurrence of the token is replaced by another value before creating the file. "%PROCID" is replaced by the current processor ID. "%LEVELID" is replaced by the ID of the fine level. For example, suppose the problem is being run on two processors and has two levels. If the filename parameter is set to "Aggregates-level%LEVELID-proc%PROCID.vtu", then the following files will be created:

- Aggregates-level0-proc-master.pvtu

- Aggregates-level0-proc0.vtu

- Aggregates-level0-proc1.vtu
- Aggregates-level0-proc-master.pvtu
- Aggregates-level1-proc0.vtu
- Aggregates-level1-proc1.vtu

These tokens prevent the vtu files from being overwritten for problems with multiple processors and/or levels. The .pvtu files can be opened directly in ParaView to display the entire level, with the aggregates from both processors.

The colormap option produces an XML file that defines a random colormap for use in ParaView. The colormap produced by this option was used to color all examples included in this article. For every few values between 0 and 1000, this file defines a random cool color - a shade of blue, green or purple. Aggregate IDs are always non-negative integers, so their colors get mapped to a cool color. Because the colormap has lots of random colors defined, aggregates with consecutive IDs end up with significantly different colors, so they contrast well with each other. Since ParaView's default colormap is a smooth gradient from blue to red, consecutive aggregates would have nearly the same color. Especially with the Point Cloud style, it was very difficult to see distinct aggregates. The negative values are reserved for graph overlays. The values -1 and -2 are mapped to red and orange, respectively. Using the custom colormap, and coloring points based on aggregate ID, normal graph edges appear red and graph edges filtered by FilteredAFactory appear orange. These lines then contrast well with the cool colors of the aggregates themselves. Figure 2.12 shows the aggregates and a fine graph with several filtered edges.

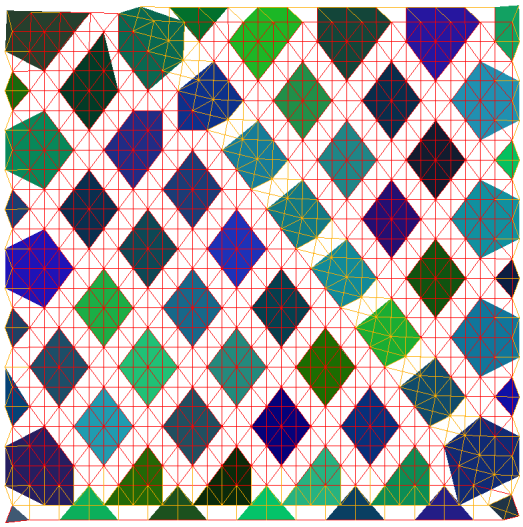


Fig. 2.12: Filtered Graph Edges: orange edges have been filtered out

3. Conclusions. The MueLu multigrid library relies on complex aggregation algorithms to coarsen sparse linear problems. The aggregates visualization project will allow both developers and users of MueLu to better understand aggregation and improve it over time. It is easy to use, requiring only two parameters to be set in MueLu's parameter list. It runs automatically within hierarchy construction and outputs native VTK, so no post-processing is necessary. The aggregates visualization should prove to be a useful tool.

REFERENCES

[1] T. Diewald, *Convex hull 3d - quickhull algorithm.* Personal Blog, March 2013.
[2] A. Prokopenko, J. Hu, T. Wiesner, C. Siefert, and R. Tuminaro, *MueLu User's Guide 1.0*, Sandia National Laboratories, 2014.
[3] T. Wiesner, M. Gee, A. Prokopenko, and J. Hu, *The MueLu Tutorial*, Sandia National Laboratories, 2014.

SIMULATING CMT-BONE COMMUNICATION ROUTINES USING LIGHT-WEIGHT NETWORK ENDPOINT MODELS

NALINI KUMAR* AND SIMON D. HAMMOND†

Abstract. In preparing to deploy application codes on next-gen supercomputers, it is vital to conduct extensive algorithm and architecture design-space exploration to identify the optimal design choices. Architecture modeling and simulation is often used to conduct such design space exploration. In this report we explore the use of high-level end-point models called 'Motifs', part of the Structural Simulation Toolkit developed at Sandia, to model and analyze the performance of key application routines in a large-scale high-performance computing code called CMT-nek.

1. Introduction. Computational scientists and engineers use High-Performance Computing (HPC) systems to simulate increasingly complex models of real world problems such as climate modeling, fluid dynamics, and nuclear reactions. Already running on massive supercomputers, more complex and detailed simulations are needed to gain insight into a particular domain, which in turn requires even faster supercomputers. Emerging and future system architectures will be able to provide the computational power necessary for these insights that scientists demand. However, these systems will likely have significantly different node and system architecture from the systems that exist today [4]. Identifying likely exascale architectures, and optimizing these applications for the candidate architectures is crucial for scientists to prepare for exascale systems.

One such application being prepared to be deployed on next-gen systems is *CMT-nek*. It is a compressible multiphase turbulence (CMT) simulation software being developed by researchers at the PSAAP-II Center for Compressible Multiphase Turbulence at University of Florida [1]. CMT-nek is being developed to perform simulation of instabilities, turbulence, and mixing in particulate-laden flows under conditions of extreme pressure and temperature. CMT has applications in many environmental, industrial, and national security areas. In many applications of national defense and security, CMT plays an important role in our ability to accurately predict and control explosive dispersal of particles.

The CMT-nek development team faces the challenging task of developing optimized software for next-gen machines whose architectures are as yet unknown and are not going to be available before the end of this decade. There are too many variables such as system architecture, programming models etc. that can affect the application performance [4]. Application code developers can benefit from early algorithm DSE before making changes to the production codes. This will require exploring application performance on various architectures via simulation.

In large-scale application codes such as CMT-nek, communication between processors and nodes is needed to ensure that any data required for computations is available locally. While computations are the useful work and communication can be thought of as an added cost or overhead. In preparing the application for a larger machine, the aim is to keep this cost in check and if possible reduce it. Network simulations can provide insight into the application performance which can then be used to alter the code to improve performance. Fine-grained network simulations, with thousands and millions of simulated end-points, can be prohibitively slow. Network simulation from application source can further increase the simuation time. Hence, for effective simulation studies we require light-weight end-point models that can be used in conjuction with a high-level application representation to conduct fast simulations.

*University of Florida, Gainesville, nkumar@hcs.ufl.edu
†Sandia National Laboratories, sdhammo@sandia.gov

The rest of the paper is organized as follows: section 2 gives an overiew of the Structural Simulation Toolkit (SST). Section 3 provides an overview of the major computation and communication operations used in CMT-nek application. In section 4 we describe the end-point models. In section 5 we describe the simulation experiments and discuss the results. Finally, section 6 summarizes the work presented in the paper.

2. Network Simulation using SST. The Structural Simulation Toolkit (SST) from Sandia National Laboratory is a parallel discrete-event simulation framework for simulation of large HPC systems at different levels of granularity [7]. SST is not a simulator by itself but a framework that provides support for integration of different simulators. The SST allows different architecture simulators and component models to be combined together to simulate a larger and more complex system. The simulators can interact and share data with each other which enables exploration of system-wide performance issues.It enables greater interoperability between simulators and models that were developed independently from each other. The ability to couple different component models is also important for fast exploration of system wide performance issues.

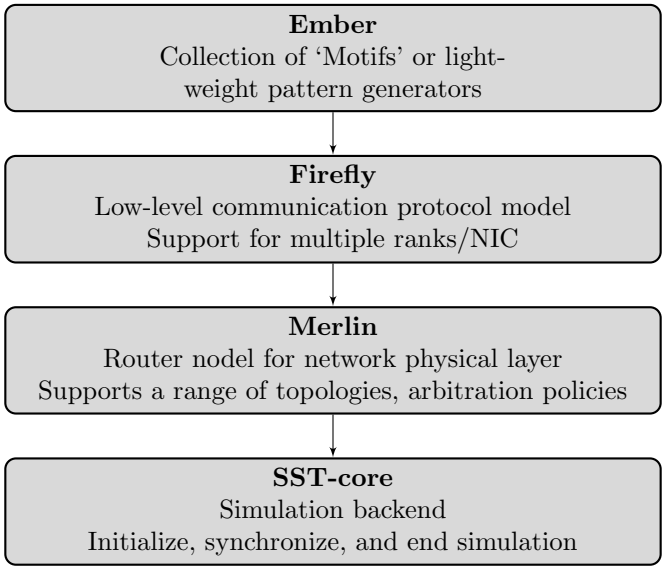
There are two main parts to SST - the Core and the Element libraries. The SST Core is the simulation backend that handles events, synchronizes execution between simulation processes, and updates the timestamps. It is responsible for setting up the simulation by instantiating components, establishing the links between these components, partitioning the simulation into MPI processes for parallel simulation, managing the causality in event queues, and ending the simulation reliably. It also provides the means for gathering statistics from a simulation run. The Core also provides other services and interfaces to the Element libraries such as a statistics API for collecting simulation statistics. The other part of SST is the Element library. Each Element in the library is a simulator or a model which interfaces with the SST Core. The Elements include processor simulators(Ariel), memory models(MemHierarchy), network models (Merlin) etc. These libraries are dynamically loaded and linked at runtime as needed for a simulation.

In this project our primary intent is to explore the performance of communication algorithms used in our application of interest, CMT-nek, as the machine and problem size grow. Typically researches use cycle-accurate simulations for architectur simulations but these simulations become prohibitively slow as the system size grows. On the other hand, analytical models can lack sufficient fidelity and depth for either system or algorithm design space exploration. For our network simulations, we use the Ember Element.

Ember is a collection of light-weight end-point models called Motifs. A Motif injects traffic into the network, receives messages, and reacts to them. As shown in Figure 2.1, Ember connects to the communication protocol model, Firefly, which in turn interfaces with the physical layer model, Merlin, that actually models the communication on the network. Since the aim of Ember is to allow simulation of systems with thousands and millions of end-points, it does not simulate processor or memory behavior. Instead, an application Motif contains a time estimate for operations that are local to an end-point.

In order to use Ember to simulate an application running on a system, the user needs to write a motif if one doesn't exist already. For large scale applications, and even mini-apps, this is a difficult task since the application behavior is to be condensed into a few lines of code in the Motif. This requires that the Motif developer be familiar with the overall flow of the application and any communication algorithms that it uses. In the next section we discuss our understanding of major phases in CMT-nek and communication algorithms unique to it.

Fig. 2.1: Network Simulation Stack in SST



3. Overview of CMT-nek. CMT-nek is a spectral element method based compressible multiphase turbulence (CMT) simulation software being developed to be deployed on exascale systems. It is being designed to hook into the Gordon Bell prize winner incompressible flow simulation code Nek5000 which has been shown to scale to a million MPI ranks [2] [8]. CMT-nek inherits functionalities such as element topology, approximation polynomials, MPI strategies, optimized linear algebra operations etc, from Nek5000 but solves fundamentally different physics equations for compressible physics as described in [6]. CMT-nek uses a domain decomposition method with generates hexahedral elements which are then mapped to cubic reference elements for the simulation. The three major phases in CMT-nek involve computing (1) the source terms, (2) the flux divergence, and (3) the numerical flux for all the cubic reference elements.

Since the application code base is massive and is under active development, it is non-trivial to understand and not conducive for fast and extensive performance profiling. To alleviate this isssue CMT-nek developers have developed a mini-app called CMT-bone which abstracts the behavior of CMT-nek [5]. We use CMT-bone as the basis for our end-point models.

As mentioned earlier, there are three phases in CMT-nek. The lastest version of CMT-nek, and hence CMT-bone, has limited multiphase coupling. As a result the **source terms** are set to zero. The **flux divergence** computation is abstracted as the dot product of the gradient operator and the flux vector. It is implemented as the multiplication of the derivative operator matrix (N, N) and the element matrix (N, N, N, Nel) to calculate the partial derivatives along the three Cartesian coordinate dimensions (r, s, t) as shown in Figure 3.1. Here N is the number of grid points along any one direction in a cubic reference element and Nel is the total number of elements in the computational domain. The elements and derivative operator matrices are fairly small, with N ranging between 5 and 25.

The **numerical flux** term is computed to enforce boundary conditions necessary to ensure continuity on the element boundaries. The flux divergence is evaluated on the surface of the reference elements which requires surface data exchange between neighboring

Table 3.1: Pseudo-code for the partial derivative calculation to estimate the flux-divergence term

Pseudo code for the partial derivative calculation of u along r performed in the spectral element solver used in CMT-nek. N is the size of each element and Nel is the number of elements per processor. Partial derivatives are computed along r, s, and t at each timestep in the simulation.
for $ie = 0$ to $Nel - 1$
for $k = 0$ to $N - 1$
for $j = 0$ to $N - 1$
for $i = 0$ to $N - 1$
for $l = 0$ to $N - 1$
$dudr(i, j, k, ie)$ += $a(i, l)$ * $u(l, j, k, ie)$

elements. These exchanges are implemented using a specialized gather-scatter library. At the start of a CMT-bone simulation, three gather-scatter methods are evaluated to determine the best performing candidate for the given machine and problem setup. These three exchange strategies are : (1) pairwise exchange, (2) crystal-router, and (3) all_reduce onto a big vector. Unlike the original Nek5000 (or its mini-app NekbonE), in all our simulations to date, we have observed that pairwise exchange is considerably faster than the crystal router and all_reduce for the problems being studied (Table 3.2).

Table 3.2: Performance of pairwise exchange and crystal router methods (per timestep)

Setup:
processes=256, processor distribution=(8,8,4), eltSize=10, total elements=25600
element distribution=(40,40,16), local element distribution=(5,5,4)

Algorithm	**Time (avg)**	**Time (min)**	**Time (max)**
Pairwise exchange	$3.189e^{-04}$s	$2.445e^{-04}$s	$3.535e^{-04}$s
Crystal router	$7.999e^{-04}$s	$7.888e^{-04}$s	$8.083e^{-04}$s

Of the two phases in CMT-bone, the flux divergence computatuion and the nearest-neighbor communication, the majority of execution time is spent in the computation part, though the cost of communication is non-negligible 3.3. In summary, the current behavior of CMT-bone can be abstracted into an end-point model or Motif with several iterations of a flux divergence computation phase followed by a nearest-neighbor exchange communication phase. We use this understanding to develop the application end-point models for study.

Table 3.3: CMT-bone profiling results for 256 ranks on 16 nodes of Compton
$nelt = 100, eltSize = 13, variable = 5$

Setup	**1 rank/core**	**2 ranks/core**
Processes	256	512
Total elements	25600	51200
Processor Distribution: px, py, pz	8,8,4	8,8,8
Element Distribution: $nelx, nely, nelz$	40,40,16	40,40,32
Local Element Distribution: mx, my, mz	5,5,4	5,5,4
Execution time metric	**1 rank/core**	**2 ranks/core**
compute time (per timestep per process)	$5.527e^{-02}$	$9.146e^{-02}$
communication time (per timestep per process)	$1.081e^{-02}$	$3.247e^{-02}$
solve time	$0.6594e^{03}$	$0.122e^{04}$
total compute time	$1.4149e^{05}$	$4.589e^{05}$
total communication time	$2.7677e^{04}$	$1.679e^{05}$

4. Developing Motifs for CMT-bone. Key application parameters that affect the workload distribution and impact performance are exposed to the user in the CMT-bone Motifs. Table 4.1 lists these parameters and their definitions.

4.1. Flux-divergence model. As can be seen from table 3.1, the application parameters that affect the execution time of flux divergence are the polynomial degree, $N-1$ and the number of elements per processor, Nel. In our motifs, $eltSize$ is the same as N and Nel is a product of (mx, my, mz). Since we are interested in the commnication performance, a detailed processor model is not required as long as we can provide a good estimate for this local operation. In our Motifs we provide three ways of providing this estimate of execution time.

First, the user can specify a fixed value for the time taken to do the flux divergence computation (derivative calculation) for all the elements of one physical quantity (velocity, presssure, etc.). Alternately, the user can provide the frequency of the processor core and the number of floating point operations per cycle that the core can compute. Based on the application parameters which specify the core workload, an estimate for computation time can be obtained. In a real system the execution time would not be the same for across different cores in the machine or across different iterations. To simulation this behavior, the user can choose to add Gaussian noise to the execution time by specifying the mean and standard deviation of the compute time. We do not have support for non-Gaussian distributions in the current models. In future, we wish to support random lookups during simulation from a lookup table containing samples of execution time obtained from real testbed experiments.

4.2. Nearest-neighbor communication model. CMT-bone contains the same implementation of MPI communication routines used in CMT-nek. The mini-app uses a simple preconditioner and assumes an ideal load balancing behavior, so the communication in CMT-bone is not exactly the same as in CMT-nek. These MPI routines - pairwise exchange and crystal router all-to-all, are part of a gather-scatter library. While the crystal_router isn't used in our limited application runs, it may be used in future as more complicated physics is implemented in the application. The spectral element coefficients are stored redundantly (and locally) on each processor instead of maintaining a global matrix and each processor

Table 4.1: Parameters for CMT3D and CMTCR motifs

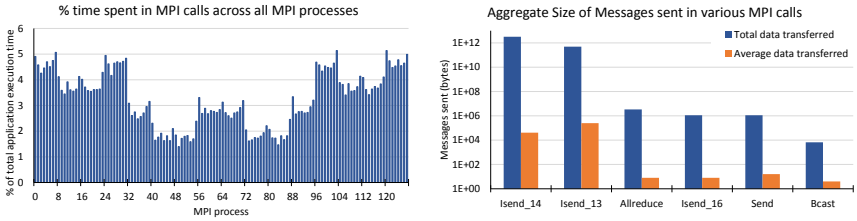
Parameter	Definition
$variable$	Number of physical properties (velocity, pressure, etc.)
$eltSize$	Size of cubic reference element (1+polynomial degree)
px, py, pz	Process decomposition in x,y,z
mx, my, mz	Element decomposition per process in x,y,z
$threads$	Threads per processor (default=1)
$procflops$	FLOPS/cycle of processor core used to estimate compute time
$procFreq$	Frequency of processor core used to estimate compute time
$nsComputeMean$	Mean for adding gaussian noise to compute time
$nsComputeStddev$	Standard deviation for adding gaussian noise to compute time
$iterations$	Timesteps or iterations of conjugate gradient solver

is given index sets containing the global ids of the elements. During a *discovery phase*, each process identifies its neighbors by identifying for every global index i on process p all the processes q that also have i. Once the neighbor lists are created, the actual transfers take place using one of the algorithms below.

4.2.1. Crystal Router. The crystal router was developed to support transfer of arbitrary length messages between arbitrary nodes on a hypercube network [3]. It performs well for applications with irregular communication behavior, where the size of transfer is known at runtime. It can be used for scalable all-to-all or all-to-many exchanges. Due to the hypercube nature of the decomposition, it guarantees termination in at most $1 + log_2 P$ stages, where P is the number of MPI processes. In our CMT-bone model, each processor communicates with it six neighboring processes. (This is unlike the Nek5000 code in which each element communicates with 26 neighbors [8]). After a process finishes its compute phase, it creates a buffer of data that it needs to send to its neighbors. During the first stage of communication every process p ($p < P/2$) sends to process $p + P/2$ all the data it needs to send to processes $P/2$ to $P - 1$. Process p ($p > P/2$) sends to process $p - P/2$ all the data it needs to send to processes 0 to $P/2 - 1$. This is done recursively for $log_2 P$ stages, where P in recursively halved with every stage.

4.2.2. Pairwise Exchange. The pairwise exchange algorithm is fairly straight-forward. After a process finishes its compute phase, it combines all the 'face' data from the element volume data into a buffer that is then transferred to its neighboring processes. The receives are blocking but the sends are non-blocking. Once a process has received all the data it needs, the iteration is complete and it can proceed to simulating the next timestep (iteration). Since each process has six neighbors, it is involved in six transfers.

5. Experiments and Results. To perform fast simulations, we had the option of using a simulator that doesn't use a central clock to synchronize events. But, application profiling results show that the approx. 90% of the communication time is spent by processors in MPI_Wait. Table 3.3 and figure 5.1(b) shows some more profiling results for CMT-bone on 16 nodes of Compton, with and without hyperthreading enabled. Compton is a 42-node ASC cluster at Sandia National Laboratories, Albuquerque, NM with two 8-core Sandy Bridge Xeon E5-2670 connected with Mellanox Infiniscale IV QDR.



(a) Time spent by each CMT-bone MPI process in communication

(b) Total and average size of messages sent in the most frequently called MPI calls

Fig. 5.1: CMT-bone pairwise exchange profiling data

6. Conclusions and Future Work. We analyzed the performance of two key CMT-bone communication routines and developed light-weight end-point models or motifs for them. These motifs were used to drive network simulations in SST to study the performance of the two communication algorithms on different architectures.

REFERENCES

[1] *Psaap-ii center for compressible multiphase turbulence.*

[2] P. F. Fischer, J. W. Lottes, and S. G. Kerkemeier, *Nek5000*, 2008.

[3] G. C. Fox, M. A. Johnson, G. A. Lyzenga, S. W. Otto, J. K. Salmon, and D. W. Walker, *Solving Problems on Concurrent Processors. Vol. 1: General Techniques and Regular Problems*, Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1988.

[4] P. Kogge, K. Bergman, S. Borkar, D. Campbell, W. Carson, W. Dally, M. Denneau, P. Franzon, W. Harrod, K. Hill, et al., *Exascale computing study: Technology challenges in achieving exascale systems*, (2008).

[5] N. Kumar, M. Sringarpure, T. Banerjee, J. Hackl, S. Balachandar, H. Lam, A. George, and S. Ranka, *Cmt-bone: A mini-app for compressible multiphase turbulence simulation software*, Workshop on Representative Applications (to appear), (2015).

[6] J. M. Powers, *Two-phase viscous modeling of compaction of granular materials*, Physics of Fluids (1994-present), 16 (2004), pp. 2975–2990.

[7] A. F. Rodrigues, K. S. Hemmert, B. W. Barrett, C. Kersey, R. Oldfield, M. Weston, R. Risen, J. Cook, P. Rosenfeld, E. CooperBalls, et al., *The structural simulation toolkit*, ACM SIGMETRICS Performance Evaluation Review, 38 (2011), pp. 37–42.

[8] T. C. Team, *The cesar codesign center: Early results*, tech. rep., April 2012.

IN SITU STREAM PROCESSING AND STORAGE FOR EXASCALE SYSTEMS

ERICH W. LOHRMANN* AND PATRICK WIDENER†

Abstract. In many systems there are often common operations that programmers will make on data. While specializing the implementation of these operations each and every time allows for extremely optimized workflows, their is often a trade off with development time and the flexibility of the solution. In this paper, we propose a general class of operator on data, which we call the *storage stone*. We explore the redevelopment of previous workflows and examine the functionality benefit that the storage stone produces in these workflows.

1. Introduction. Scientific applications are often built at the highest level of abstraction, by scientists to study their particular area of science. These scientists often have a limited, if any, knowledge of the inner mechanisms of computers. However, some problems that researchers in non Computer Science fields face often require the use of computers. While small scale projects can be done with little difficulty by amateur computer programmers, large scale projects sometimes require intimate knowledge of computers and the way in which they allocate resources. For this reason, some computer scientists build tools that work on the system level to improve resource allocation across High Performance Computing. When these tools are libraries that are used to build complex systems, we call them "middleware". Middleware is often developed to make it easier for scientists to build high performance applications, without worrying about the underlying mechanisms necessary for high performance.

In this paper, we will focus on extensions to a communication middleware known as EVPath. EVPath is a high performance communication middleware used to transport arbitrary data types between arbitrary nodes with arbitrary byte order and arbitrary hardware. It also does this very, very quickly. EVPath is not a communication system. EVPath is the library tool used to build many different and versatile types of high performance communication systems. Additionally, EVPath contains pieces that naturally support the concept of high performance application components.

In the world of high performance computing, complete applications are often broken down into several different components. These "workflows" often consist of more than one process that communicate with each other using some type of high performance communication system. These workflows are often highly specialized to meet the needs of the input data, the physical hardware the workflow runs on, and the format of the output data. However, while the system may be specialized, there do exist some components of the workflow that are general enough to be reusable.

For example, it is often true that a mechanism for buffering data, for analysis and to guarantee redundancy, is often needed between components of the workflow. At other times, data must be gathered together to run some algorithm. This is typically done, using either a "sliding window" or "tumbling window" algorithm, both of which are explored below.

I propose that these are all examples of the same general class of operator on data. This class of operators is prevalent in many different types of systems, from monitoring to high performance computing applications. This work is an extension of EVPath[2], which is in essence a high performance data transport system with flexible data typing. The EVPath library stack includes the ability to send filtering and transformation code from the receiving process to the source process, all while compiling the code on the source machine to ensure best use of heterogeneous software.

*Georgia Tech College of Computing, ErichLohrmann@gmail.com
†Sandia National Laboratories, pwidener@sandia.gov

The transported and compiled code is known as CoD or C on Demand[3]. EVPath's API is filled with many different "stones". Each "stone" has a basic purpose, such as transforming, filtering, sending, or receiving data. The EVPath stones have a natural use in the development of the generic operations mentioned above. The stones that transform and filter data on a node aside from the one who originates the source, write their code in CoD, which is a small subset of C. We believe that while comprehensive the list of necessary stones is not yet complete. For instance, we have developed a "storage stone" which is actually a class of more specific stones with related but different purposes.

The storage stone, provides aggregation and redundancy functionality to the EVPath library. This functionality is captured in separate subclasses of the generic storage stone. The first is the redundancy storage stone, which is in essence actually two stones, a store storage stone and a transaction storage stone. The store storage stone acts in the flow graph where we wish for data to be stored with some guarantee of persistence (though not necessarily non-volatile persistence). The transaction storage stone is used by the system to guarantee that the data is received on the far end of the workflow. In this way, we have created a generic form of "streaming persistence" in workflows that transport data to potentially many different nodes.

In addition to the redundancy functionality of the storage stone, this class of stone also carries the functionality of aggregation. In particular we implement the logic for a "sliding window" and a "tumbling window" stone.

A "sliding window" of data is a data set with n, number of elements. These n elements make up a single complete window. Some operation, typically an average or some other mathematical operation, operates on the window. In a sliding window, when a new element enters a complete window, the oldest element is removed. Therefore, all complete windows have n elements. On the other hand, a rolling window acts differently when a new element is introduced to a full window. A complete rolling window still has n elements, but when a new element enters the complete window, all current elements in the window is discarded. Rolling windows normally never act on any incomplete windows, so operations almost always wait until the window is once again complete. The difference between the two windows is most easily seen by the number of operations on the data. A rolling window will only act on its data set every n iterations of data and a sliding window acts every iteration of data after the first n iterations make a complete window.

These stones are single stones that can be used to control throughput of the data that runs through the network. In addition, it allows for the quick processing of data when a specified amount of data arrives at the storage stone. All storage stones share the same characteristic that they act on some subset of the data.

In this paper, we examine the benefits of the storage stone in two different already implemented systems. The first is the Proactive Directory Service [1], or PDS, which is a directory service that allows applications to register a callback handler to be notified when some element registered in PDS changes. The second is the SmartPointer [4], which is an multiprocess scientific workflow system.

1.1. EVPath. EVPath is a communication middleware library. EVPath, as previously mentioned, is not a communication system. The differences are subtle, but important. A communication system is the layer on which an application or a group of applications leverage to move data around on the system. EVPath, on the other hand, is a library used to build that communication system. As such, EVPath's API is filled with a series of "stones" that manipulate or transport data in a particular manner. A typical EVPath system consists of several nodes, all with one or more stones on each node. EVPath is the highest level of abstraction, built on top of other library layers that are out of the scope of

this paper, but are worth mentioning as they provide some of the functionality described above.

Each stone in EVPath, typically contains one action. These actions divide the stones into separate stone "types". For instance, an action that filters out unwanted data is known as a "filter stone". In addition to the filter stone, their exist: transform stones, source stones, sink stones, bridge stones, router stones, split stones, and multiqueue stones. All of the stones are named canonically, therefore transform stones transform the data, filter stones filter the data, etc.

Individual stones and nodes are not aware of the layout of the entire data flow graph of a specific application or groups of applications. Instead, when specificity is needed in certain stones, stones subscribe to a certain data type and all other data types are discarded. Furthermore, some stones like the router stone, split stone, and multiqueue stone, can publish to multiple different stones. As you can see, EVPath stones are the "fundamental" types that many communication systems need to build their systems. In fact, our ability to place the stones on any node, without regard for the total data flow graph, allows us more flexibility in where data is processed and even has the added feature that upgrades to the system can be piecewise, node by node.

2. The Storage Stone.

2.1. Background Information. The storage stone is a natural extension of the EVPath library. The EVPath library stack evolved out of a need for easy development of high performance transport systems that worked well across heterogeneous systems and with heterogeneous data. We reiterate that EVPath is not a transport system itself, but rather a library that manages a lot of the nasty communication code that inevitably must be written as a consequence of building a high level transport system.

With that design goal in mind, the API consists of creating and controlling a series of "stones". For instance, EVPath will allow you to create a "bridge stone" which is a stone that sets up the connection to transport data across the network from a sending node to a receiving node. APIs for creating "filter stones" and "transform stones" also exist, both of which perform operations on the data exactly as their names imply. Each stone lives in the node and process in which it is created. Furthermore, the EVPath library makes interprocess data transport on the same node, memory and computation efficient. Data type definitions in EVPath are handled by a specific type definition system known as FFS. The unique power of FFS allows EVPath to operate on expanded types. Since stones are per node per process and FFS allows stones to pick up expanded types, upgrades to the system can be done piecewise rather than taking down and upgrading the system all at once.

While the storage stones that implement the windows can live in the base EVPath, the redundant focused storage stones that provide the "streaming persistence" require connection information from "upstream". As described previously, EVPath does not naturally provide this information as data flows into stones set up and connected on a node. Luckily there is a small layer built on top of EVPath known as EVdfg. EVdfg allows a master data flow graph process to control the connections between processes that register with the master dfg process. This allows for the natural extension of the persistent storage stones as the upstream information is available to the master process.

2.2. Technical Details. It is in this environment that the storage stone has been created. The storage stone is implemented on top of a previous stone known as a multiqueue stone. The multiqueue stone will store a data type in a couple of queues depending on its data type and then run a CoD based handler function. It is possible for the multiqueue stone to submit data to multiple different ports and therefore multiple different pathways

in the network data flow. Naturally, as the storage stone is implemented on top of the multiqueue stone it inherits all of this functionality.

Each type of storage stone has a slight different implementation on top of the multiqueue stone. For the window storage stones, it is easy to implement a piece of CoD code that is inserted at the beginning and the end of the user provided CoD code that facilitates the different window like behavior. In this way, the storage stone user provided code is run only when there is a complete window and the stone can output any type of user defined data type.

On the other hand, the persistent stone is a little harder to produce. Two stones must technically be set up, the store storage stone and the transaction storage stone. Luckily, the transaction stone requires no CoD code to be written by the user. However, a store storage stone must listen to some data from the transaction stone associated with the sink stone on the farthest end of the data flow graph. The underlying messaging structure is fairly generic, so the user need only set up a transaction stone in the point of the data flow graph that he wishes to guarantee persistence.

The transaction stones main job is to keep a list of the messages it should see, versus the id of the messages it actually receives. The transaction stone must keep an individual integer tally because as in any distributed system the ordering of events can be fuzzy. Should a sink stone request a transaction check and the transaction stone has not received all of the intended messages, the transaction stone sends a resend data type message to the store storage stone. The store storage stone then sends the messages down a different dfg path in an attempt to resend the messages. The success of this attempt can vary depending on the extent of the failure and the nature of the dfg.

How we determine which path to send the data down after a failure and how many dfg options exist are system dependent. We have built the storage stone to handle multiple output paths, but the user must always determine the policy mechanisms for when a failure occurs. Additionally, there is an option to write the data out to disk, which may be used in handling repeated failures or may be used as the primary means of storage rather than having the data live in memory. This may free up RAM space overhead in a trade off for time, or it may be useful when dealing with deep memory hierarchies and fast non-volatile memory.

3. Use Cases.

3.1. Proactive Directory Service. In some systems, there exists a need to watch some specific data for change. These applications are known as state full applications and they are dependent upon the state of some data. This data may not even necessarily exist on node. A naive solution to such a problem is to continually poll the data at some fixed or varied point in execution to see if the state has changed. It is easy to see that such an approach does not scale well when many applications need to poll many different data states.

PDS attempts to address this basic limitation by instituting a Proactive approach to the problem. A client application can register a handler with the server PDS for some data structure. When that data structure changes, the server will then push the event change to all clients that have registered callback functions. This obviously reduces network traffic as well as wasted CPU cycles created by the previous continuous polling approach. Furthermore, it was possible to filter data in such a way so that only the relevant messages reached the interested client. The filtering was done on the server side, to reduce unnecessary network traffic.

It is in this environment that we wanted to introduce aggregate storage stones. The idea being that we could not only filter the data, but potentially get an average or some

other mathematical comparison on pieces of the outgoing data and output the aggregated value. This would allow the client more control over the evaluation of the data.

3.2. Technical Details of the PDS Implementation. The original PDS code is old, near ancient with respect to the modern library stack, so the first part of my work was to renew the code. The storage stone classes do not need any special new EVPath APIs, but the PDS code was written in a time when the EVPath library was not fully realized. The old PDS code did not fully utilize the multiple type functionality. More specifically, the old PDS code used a single data structure to relay multiple changes to a single event type. To do this, the authors of the code created a general blob char pointer payload entity inside of the event type. This in itself, was not a problem in the normal C handlers as C has the ability to pack and unpack data easily using creative type pointers.

Unfortunately, CoD does not have this ability. Since the functionality of the storage stones are written in CoD, we needed the ability to not only look at the data, but manipulate and change it in situ. Since CoD could not handle the placing and renaming of pointer payloads correctly, we were forced to fundamentally change the structure of PDS to accommodate multiple specific types for data objects. This was a fairly large rewrite as we wanted to ensure that PDS maintained its original functionality in addition to the new storage stone functionality. This took some time as PDS allows users to register for handlers to data structures that have not yet been created.

The reason this rearchitecting was necessary was to facilitate the use of the storage stone for aggregation operations on some data structure in a natural manner. To demonstrate this functionality, we are aiming to use PDS to monitor cluster system information and make intelligent decisions based on averages or other mathematical summations across small data sets. By implementing both the sliding window and the rolling window, we can easily monitor the average output of data on the node itself in situ, rather than ship the network data to some centralized or distributed client. We hope to prove reduced network traffic with similar if not less lines of code.

3.3. SmartPointer. The SmartPointer system is a comprehensive scientific data workflow that analyzes the output from a LAAMPS simulation. The data is all of a block of material, modeled with atoms, that is put under pressure and breaks in the relative middle of the simulation. It is a complicated system with many separate processes, all of which can be run on separate nodes or the same node, thanks to the EVPath transport mechanism. Please refer to Figure 3.1 for a diagram on the separate components of the SmartPointer system.

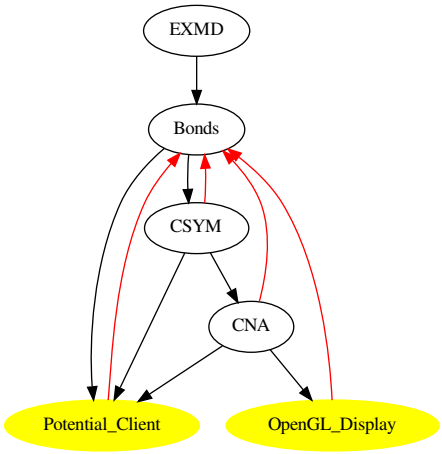


Fig. 3.1: The SmartPointer workflow diagram. The red arrows indicate feedback loops that allow the source process to change the bond calculations. The yellow filled circles are processes that connect to the workflow at runtime and can subscribe to any specific channel. In this case, the OpenGL display only subscribes to the CNA channel, while the Potential_Client subscribes to the Bond Channel, CNA channel and CSYM channel.

Aside from the output LAAMPS simulation, their exists a bond server, which efficiently writes out which atoms are bonded to which other atoms with respect to two radius values. The first radius value determines the minimum distance that an atom can be away and still be bonded, and the second determines a maximum distance that an atom can be away from an atom and still be bonded. In addition, the bond server also outputs radial distribution data that details the distance between atoms partitioned into some n, number of bins, and set to only look at atoms below some max radial difference of rmax.

The bond server emits the radial distribution directly to any interested client. It then emits the bond information to another piece of the system known as the CSym processes. This process uses the bond information to calculate a central symmetry number for each atom and adds it to the data structure. The central symmetry value can be used by display components as a convenient tag to find pieces that are close to the breaking point. This value is not currently used by any real display components but it is part of the system so we felt we should mention it.

The CSym process then emits to a process called CNA. CNA stands for common neighbor analysis. The common neighbor analysis process takes the atoms and their associated bonds and analyzes their connections to give each atom a "color". This color allows us to easily visualize common patterns on the final display pieces if we so choose. There is some difficulty here with the bond distances mentioned in the bond server. We will explain in detail when we get to that piece in the next section.

Finally, their exists a display component that subscribes to the output of the CNA server. While it is possible, and in fact a feature, of the SmartPointer system to allow multiple clients to register to different parts of the system to cut down on the associated data, we only have the one display component currently running. It depicts the block of

atoms literally, each with their own color. It utilizes deprecated fixed-processing pipeline OpenGL code to depict the atoms and their colors, but it is good enough to view the simulation with no visible delay.

The entire SmartPointer system was tied together with EVPath. EVPath manages the data being sent from one place to another and it allows for the "channels" of data to have sources but no sinks. This is useful functionality as it allows us to set up "channels" that could be subscribed to in the future without requiring that the data must go somewhere in the beginning. Furthermore, EVPath allows for a feedback loop between the display components and the bond server. To find the correct min and max radius bonding values between atoms, we need the radial distribution of the data and have the ability to change it on the fly. It is for this reason that the bond server outputs the radial distribution data. The bond server then subscribes to a feedback "channel" from the display components, so that it can bond data differently as time moves forward.

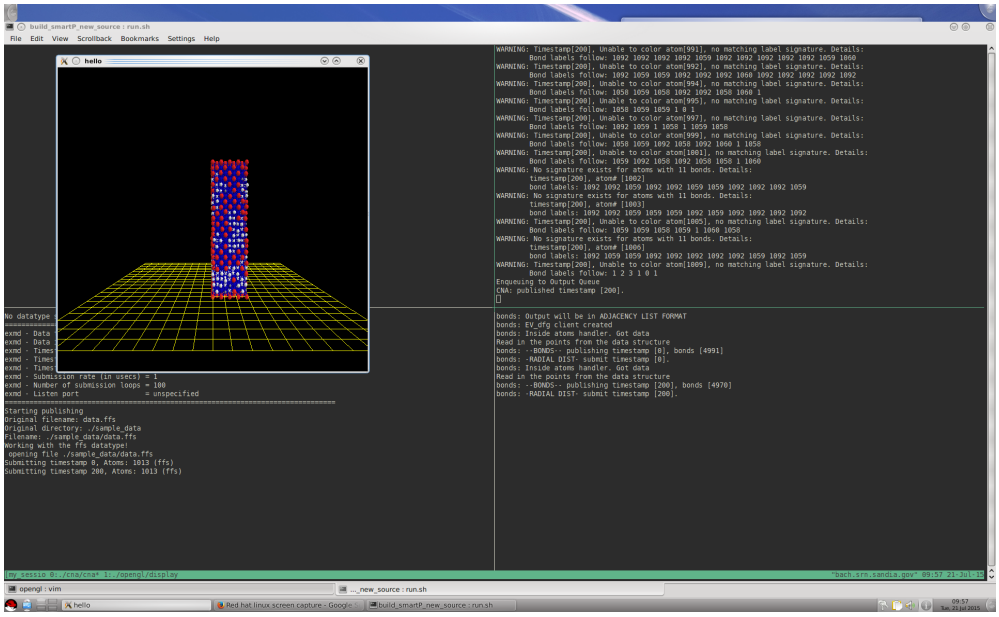


Fig. 3.2: SmartPointer system example output

3.4. Technical Details. The original SmartPointer system was written approximately a decade ago. It was originally written for the use of Ipaq clients. Since then, some work has been done to fix Android clients and utilize CUDA in part of the system to speed computations. As you can imagine the SVN directory where the code lived was a complete mess. We worked to clean up the folder and reintroduce only the most essential parts to our system, stealing and reusing code where we were able to do so. However, their were some key missing pieces that had to be replaces.

For instance, the bond server used a mathematical C library to create a KDtree for the atoms. The KDtree was then supposed to speed computation when searching for bonds. Unfortunately, this library has changed/was no where to be found and so we had to reimplement the KDtree search of the bonds. We chose to replace the libraries functionality with an open source C++ data structure known as KDtree++. While the code to create

the KDtree already implemented in the open source code, it took us some time to invent our own algorithm to do the bond searching with the two radius's quickly enough to beat a straight linear search. This detour took some time.

Furthermore, while the EVPath code was not deprecated, we decided that it would position us best for future research if we moved away from the old peer-to-peer model to a more centralized model. EVPath has a thin layer on top that makes this fairly easy, known as EVdfg. This transition took some time to facilitate as the communication mechanisms of all of the components had to be switched over to the new model.

In one of the iterations of the SmartPointer system, there was an attempt to utilize CUDA and the GPU for some auto-correlation functions over a subset of the data. These functions implemented some sliding window and rolling window functionality. The goal of getting the SmartPointer system updated is to reuse these functions but implement them with the storage stones. The storage stones do not add any new functionality, but rather are simply replacing the same general code with a general interface to demonstrate the usefulness and practicality of the storage stone itself.

4. Conclusion and Future Work. In conclusion, the majority of the work produced thus far has been in revitalizing and reworking old code to allow us to add functionality to the existing frameworks. Within the next couple of weeks, we hope to have running experiments detailing the usefulness of the Storage stone in two separate systems. The revitalization of the SmartPointer workflow is not yet complete, as we still need to add a component that will allow for clients to subscribe to a specific channel in the workflow. This mechanism was lost when we switched over to the centralized model, but it is possible to add it back with a little more time.

Furthermore, the Storage Stone will find its way into the SmartPointer workflow by moving some of the CUDA mathematics code into the bond server directly. This too must be done in the next couple of weeks. Our goal is to write a paper detailing the in situ benefits of the Storage stone in systems like these.

REFERENCES

[1] F. E. Bustamante, P. Widener, and K. Schwan, *Scalable directory services using proactivity*, IEE/ACM SC Conference, 2002.

[2] G. Eisenhauer, M. Wolf, H. Abassi, and K. Schwan, *Event-based systems: opportunities and challenges at exascale*, 2009.

[3] G. Eisenhauer, M. Wolf, H. Abbassi, S. Klasky, and K. Schwan, *A type system for high performance communication and computation*, 2011.

[4] M. Wolf, Z. Cai, W. Huang, and K. Schwan, *Smartpointers: Personalized scientific data pointers in your hand*, 2002.

MATERIAL MODELS FOR NEXT GENERATION PLATFORMS

NICOLAS MORALES*, DAVID LITTLEWOOD†, AND STAN MOORE‡

Abstract. Compute clusters for scientific computing platforms are beginning to take advantage of GPU accelerators and coprocessors like the Intel Xeon Phi that provide compute capability far beyond what a single CPU node can achieve. We aim to understand how these next generation computing platforms can be used to improve the performance of solid mechanics codes through an investigation of the scientific computing package Albany's Laboratory for Computational Mechanics (LCM).

Specifically, we examine the impact material model evaluations have on the performance of the overall solution process. We provide a new material model implementation that uses the Kokkos library for cross-platform support on nodes with accelerators or coprocessors. We suggest modifications to the LCM material models, Albany, and the underlying Trilinos library that Albany uses, including Kokkos.

Our results show that more complex material models can have a significant impact on solve time. We show that even with simple material models we can improve the performance of the constitutive model kernel for large models (100000 elements) by using accelerators. Finally, we provide a framework for the path forward for taking advantage of these next generation computing platforms in Sandia's engineering analysis codes.

1. Introduction. The use of accelerators and coprocessors is becoming more prevalent in the world of high performance computing and scientific computing [4, 5]. Future Next Generation Platform (NGP) supercomputers planned by the Department of Energy will utilize accelerators to provide peak compute performance. Sandia's production software does not currently take advantage of these hybrid clusters.

One goal of the Albany framework is to provide a proving ground for new techniques in scientific computing and differential equation solvers [6]. The target of our work is to evaluate the difficulty and utility of implementing parallel NGP code for solid mechanics simulations. This can be achieved by first implementing the code modifications in the solid mechanics module of Albany (called LCM) and using it as a test bed. Eventually, optimizations used in Albany can be reimplemented in production codes such as Sierra [8] or the LAME constitutive models [7].

We target the material models in Albany because of both the simplicity and relative isolation of the material code compared to other parts of Albany and because of the potential performance gains that can be achieved by optimizing the material model code. Initial measurements show that especially for more complicated material models such as the Crystal Plasticity model, evaluation time for the materials can consume roughly 72% of the total Albany execution time. This implies that optimizing the material model can create significant speedups in Albany simulations.

We use the Kokkos library, found in the scientific computing package Trilinos, as the platform for this parallelization. Kokkos is a parallelization library that uses Views to abstract the way in which data is used by the application. This provides a great amount of portability between CPUs, accelerators, and coprocessors [2, 3]. Ideally, any Kokkos code can run on any of these platforms. Previous uses of Kokkos in Albany have focused on parallelizing finite element evaluations for ice sheet simulations [1].

Lessons learned through the application of Kokkos to constitutive models in Albany will provide insight on the adaptation of Sandia's production engineering codes for emerging next-generation platforms.

*UNC Chapel Hill, nmorales@cs.unc.edu
†Sandia National Laboratories, djlittl@sandia.gov
‡Sandia National Laboratories, stamoor@sandia.gov

2. Implementation. We provide a new material model interface in Albany/LCM, called `ParallelConstitutiveModel` (Listing 1). This interface provides a way for users to provide an evaluation kernel that is run for each element. The evaluation kernel must be provided as a functor (with `operator()` defined) or as a C++11 lambda.

Listing 1: The base class interface for parallel constitutive models in Albany/LCM.

```
template<typename EvalT, typename Traits, typename Kernel>
class ParallelConstitutiveModel: public ConstitutiveModel<EvalT, Traits>
{
  // ...

protected:

  virtual Kernel createEvalKernel( FieldMap &dep_fields, FieldMap &eval_fields,
       int numCells) = 0;
};
```

The pure virtual `createEvalKernel()` method must be implemented by any inheriting class. This function is called by the base class before any material model evaluation, and constructs and initializes any internal data of the kernel (such as dependent fields). The type of this functor is passed in to the interface as the third template argument which allows the base class to use the type in the internal Kokkos `parallel_for` call. The kernel code itself, supplied either by a lambda or by `operator()` on the functor must be declared `KOKKOS_INLINE_FUNCTION`.

2.1. Parallelization. `ParallelConstitutiveModel` uses the Kokkos library in the Trilinos packages in order to parallelize the material model kernels. The advantages of using Kokkos are that code can be written similarly for all platforms, including GPU accelerators or CPU-like coprocessors. However, a significant caveat of this is that kernel code cannot refer to non-kernel code. This means that every function or method called from kernel code must itself be declared `KOKKOS_INLINE_FUNCTION`. This means that STL functions cannot be called from inside the kernel and neither can standard library math functions. Any third party code must be modified to work inside a Kokkos kernel.

This is a significant but not insurmountable limitation. Most of the material models in LCM have a limited set of external code that they refer to. For example, in order to replace the Neohookean material model with parallel code, we only had to implement a Kokkos compatible tensor class. However, other models such as the Crystal Plasticity model would require more complex tasks, including the implementation of a nonlinear solver in Kokkos. Furthermore, some models use the Sacado automatic differentiation library that is part of Trilinos. In order for the parallel constitutive model to be able to use automatic differentiation, Sacado must support Kokkos.

Other problems are related to the dynamic nature of the Albany library. Most of the Albany code is generalized in order to provide a flexible interface for experimentation in computational simulation. This can have performance implications. For example, Albany supports a run-time dependent dimensionality. This dimensionality is used to determine the allocation sizes of many classes, including tensors that are used locally in the kernel. While CUDA and Kokkos support the use of `operator new()` in kernel code, this internally calls the GPU heap allocator. In smaller kernels such as the Neohookean material model, the use of dynamic memory allocation can have severe performance implications. Because of this, we limited the tensor code used in the parallel material models to three dimensions. Support could be added for higher-dimensional tensors using the heap allocator for any dimension above 3 while maintaining the use of fixed stack array for dimensions less than or equal to

3.

The parallel kernel is instantiated once per material and applied to each element that uses the material. Internally, each element must also iterate over all of the quadrature points for that element. We decided not to parallelize this inner loop because the number of integration points is always relatively small, and for the simpler material models there would be little benefit for doing so. Each inner loop iteration of a Neohookean kernel, for example, is simple enough to be executed in a short enough amount of time for the transfer time to dominate even more on the GPU.

Data required by the kernel such as the Jacobian or the deformation gradient is stored using the existing `MDField` class in the Phalanx package of Trilinos. This class internally uses a Kokkos view to represent the data on device code, allowing much of the existing code to remain unchanged. The data on GPUs is set using Unified Virtual Memory (UVM). This makes the implementation simpler at the cost of a slight performance hit. We also implemented code which allows this data to be transferred using pinned/pageable memory for better performance. This is done using Kokkos host mirrors and deep copies.

2.2. Extensions to Albany and Other Contributions. In addition to the parallel constitutive model interface, we implemented a parallel version of the Neohookean material model. This can be used to speed up simulations that use the material model in addition to being used as an example for parallelizations of more constitutive models.

In order to measure the results of the parallelization, several performance monitor classes were implemented. These classes include the `CounterMonitor` object, which tracks several counters that can be implemented or decremented; `TimerMonitor` which tracks the accumulated time in a set of timers; and `VariableMonitor` which tracks the history of a variable as its value changes over time. All of these monitors output CSV table data that can be read by most spreadsheet programs or parsed by post-processing scripts. We also implemented a minimal generic tensor class that closely mirrors the functionality of the existing Intrepid MiniTensor in Trilinos. However, the tensor class uses stack instead of heap memory for performance reasons and can also be used in Kokkos kernel code. Finally, we added support in the Material Point Simulator for multiple elements for performance testing.

Other contributions include a wiki entry and scripts for Kokkos support in Albany on the Shannon compute cluster. We also implemented a set of post-processing python scripts that can generate plots and graphs from a set of CSV files representing different runs of Albany or the Material Point Simulator.

3. Results. We tested the parallel constitutive model code on the Shannon testbed cluster. Each node of the Shannon testbed has two 8-core Sandy Bridge Xeon E5-2670s running at 2.6GHz with hyperthreading deactivated. The nodes we used each had 4 NVIDIA k40m GPUs. In all our results, we used only one CPU core and a single GPU.

All our tests were run on a modified version of the Material Point Simulator. This Albany tool is used to test new constitutive models and does minimal computation outside of the material models. The Material Point Simulator normally works on a single element and quadrature point, but we modified it to take a variable number of elements, defaulting to one. Each element is identical with the same operation being repeated on all the elements. This allowed us to measure how the parallel algorithm scales with the number of elements in the simulation.

We used the Neohookean material model for the experiments. The model was simple to convert to parallel code, only needing modification to the tensor class used for the material model evaluation. The Neohookean model is fast and efficient to evaluate, requiring few floating point operations to execute and not needing any internal iterative solvers or other

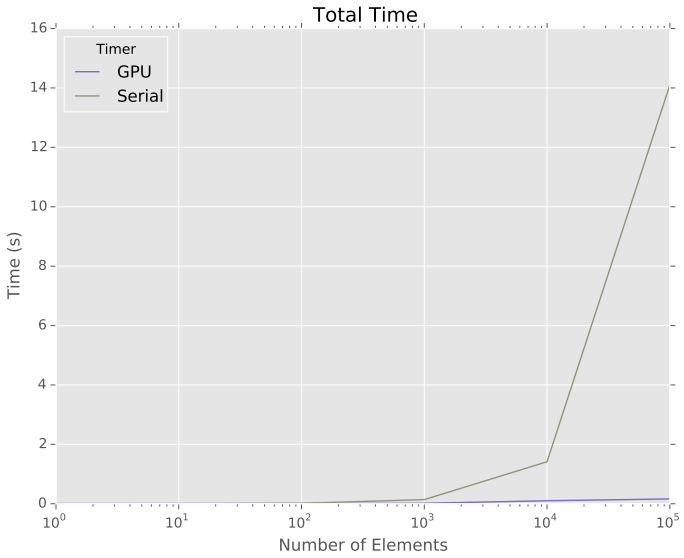


Fig. 3.1: Total time taken by the serial and GPU versions of the Material Point Simulator.

similar complex operations. This allows us to have a baseline material model that does not vary in execution time depending on the input to the model.

The first performance analysis was done comparing the overall computation time of a single-threaded CPU version compared to the GPU version using one GPU. The number of elements ranged from 1 element to 100000 elements with one quadrature point per element. For less than 100 elements the serial version performed better due to the overhead of transferring data to the GPU and synchronizing results with the CPU. However, as the number of elements increased, the GPU started to dramatically outperform the serial version (Figure 3.1).

The second experiment measured the timing of different aspects of the simulation, including the transfer time of the data needed by the kernel and the actual execution time of the kernel. Figure 3.2 shows the different timings. The transfer time increases linearly with the number of elements. However, the execution time does not. Therefore the ratio of transfer time to execution time increases as the number of elements increases.

This ratio becomes proportionately less as the complexity of the material model increases since the ratio of bytes transferred to floating point operations executed decreases. Furthermore, the transfer time can be mitigated by overlapping the communication with some computation. If the CPU can do work while data is being transferred, the transfer time is effectively hidden. Finally, if more of Albany/LCM is parallelized using Kokkos, some or all of the input data to the material model could already exist on device memory. This would mean that it would not be necessary to transfer this data for evaluation of the constitutive model.

The third experiment investigates the transfer times with CUDA Unified Virtual Memory (UVM) compared to using pinned/pageable memory. Overall, there is approximately a 16% performance gain in using pinned/pageable memory. This is a performance tradeoff between ease of programming (UVM) and performance (pinned/pageable memory). The

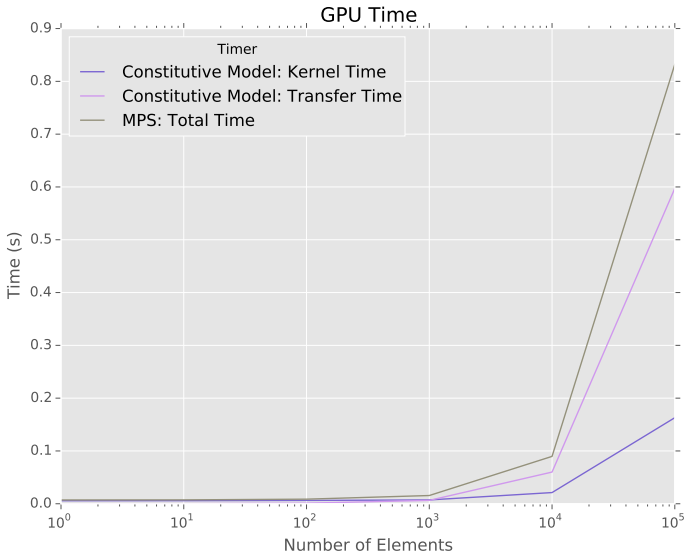


Fig. 3.2: Time taken by the kernel execution and data transfer, compared to the total time taken by the GPU version of the Material Point Simulator.

performance gains only matter when data transfer is a significant portion of execution time. With more complicated models, this ratio becomes less and bandwidth savings become less effective.

Additionally, specifically optimizing for GPU memory transfers in this way introduces an extra copy operation that is not necessary on the CPU version of the code. Although Kokkos abstracts away some of the differences between host and device code, some optimizations require specific knowledge of what execution space Kokkos is actually running in.

These results show a significant performance advantage in parallelizing Albany/LCM's constitutive material models. This is especially significant because the Neohookean model can be evaluated quickly, making it potentially less feasible for use on the GPU because the transfer time is comparatively large. It is expected that for more complex material models, the performance gains will be greater, although the required code modifications will increase in complexity.

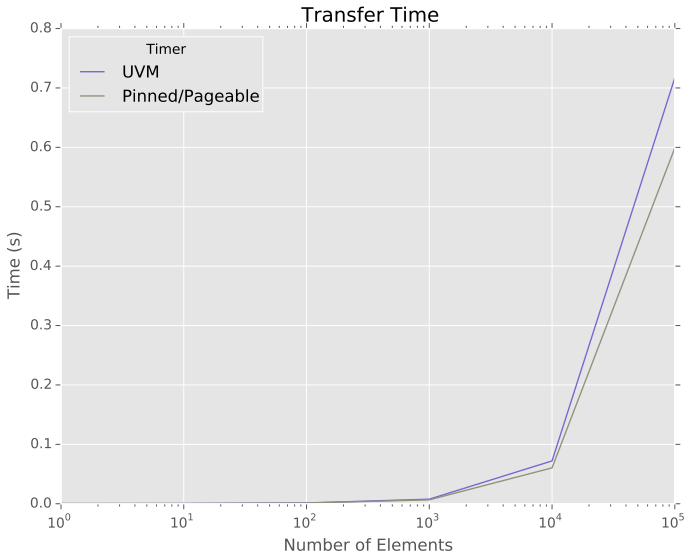


Fig. 3.3: Transfer time comparison between Unified Virtual Memory (UVM) and pinned/pageable memory on the GPU version of the Material Point Simulator.

4. Conclusion. Our parallel implementation of the Albany/LCM constitutive material model shows significant performance advantages over serialized code. These results show the way forward for parallelizing different aspects of Albany or production-ready computational mechanics codes. We demonstrate a path forward with some caveats to be aware of.

First, we have to deal with the problem of transfer time to the GPU. Three solutions were proposed: using more complicated material models, overlapping communication with computation, and parallelizing more parts of Kokkos so that the input to the material models already exists on the device.

Secondly, we brought up an issue with the dynamic nature of Albany. The goal of Albany is to be general purpose; however, this can cause some performance inefficiencies in device code where heap memory allocation is expensive. Furthermore, the memory that the internal CUDA implementation can use for the heap is fixed for each kernel run. This removes some of the abstraction that Kokkos uses.

Thirdly, the parallelization of more material models or of other aspects of Albany/LCM requires that certain primitives be implemented in Kokkos, due to the dual host/device nature of Kokkos. This means that C++ STL algorithmic primitives, math code such as matrices or tensors, and solvers must have a Kokkos-friendly implementation. The advantage is that this code can also be re-used for non-Kokkos code, with minimal performance impact.

As we move forward with implementing our testing and production codes like Sierra or LAME for use in next generation platforms, it is important that all of these caveats be considered. Targets for parallelization should be chosen because of their performance characteristics and the ease of porting them to next generation computing platforms.

REFERENCES

[1] I. Demeshko, H. C. Edwards, M. A. Heroux, R. P. Pawlowski, E. T. Phipps, and A. G. Salinger, *Kokkos implementation of Albany: a performance-portable finite element application.*

[2] H. C. Edwards and D. Sunderland, *Kokkos array performance-portable manycore programming model*, in Proceedings of the 2012 International Workshop on Programming Models and Applications for Multicores and Manycores, ACM, 2012, pp. 1–10.

[3] H. C. Edwards, C. R. Trott, and D. Sunderland, *Kokkos: Enabling manycore performance portability through polymorphic memory access patterns*, Journal of Parallel and Distributed Computing, 74 (2014), pp. 3202–3216.

[4] V. V. Kindratenko, J. J. Enos, G. Shi, M. T. Showerman, G. W. Arnold, J. E. Stone, J. C. Phillips, and W.-m. Hwu, *GPU clusters for high-performance computing*, in Cluster Computing and Workshops, 2009. CLUSTER'09. IEEE International Conference on, IEEE, 2009, pp. 1–8.

[5] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell, *A survey of general-purpose computation on graphics hardware*, in Computer graphics forum, vol. 26, Wiley Online Library, 2007, pp. 80–113.

[6] A. G. Salinger, R. A. Bartett, Q. Chen, X. Gao, G. Hansen, I. Kalashnikova, A. Mota, R. P. Muller, E. Nielsen, J. Ostien, et al., *Albany: A component-based partial differential equation code built on trilinos.*, tech. rep., Sandia National Laboratories Livermore, CA; Sandia National Laboratories (SNL-NM), Albuquerque, NM (United States), 2013.

[7] W. M. Scherzinger and D. C. Hammerand, *Constitutive models in LAME*, SAND Report 2007-5873, Sandia National Laboratories, Albuquerque, NM and Livermore, CA, 2007.

[8] SIERRA Solid Mechanics Team, *Sierra/SolidMechanics 4.36 user's guide*, SAND Report 2015-2199, Sandia National Laboratories, Albuquerque, NM and Livermore, CA, 2015.

MARCHING CUBES APPLICATION FOR MANTEVO

STEVEN J. MUNN* AND KENNETH MORELAND†

Abstract. We propose a set of highly portable, lightweight, and scalable implementations of the Marching Cubes algorithm in C++ allowing users to easily modify and optimize the software for testing and benchmarking purposes. Our application fits within the Mantevo project guidelines to encourage widespread usage in the high-performance computing community. It includes implementations of Marching Cubes in serial and in parallel using OpenMP and MPI.

1. Introduction. Large-scale scientific visualization tasks are often time sensitive and computationally expensive. Industry standard software in the field is typically large and sophisticated making it difficult to optimize for new hardware. The core idea driving the Mantevo project is to break down large programs into minimalistic key components [1], which are easy to optimize on any system including hardware prototypes. In this spirit, we focus our attention on a central component of scientific visualization: Marching Cubes.

Cline *et al.* first proposed this algorithm in 1987 [6] and it is widely used in standard software today such as ParaView or VTK from Kitware [2] [4]. It is a simple procedure, but choosing the best hardware and implementation algorithm for it can be diffiucult. Specifically, when speed or volume constraints in modern scientific applications warrant a parallel implementation of the technique, the amount of effective parallelism available depends on the size of the problem and the type of hardware on hand. These constraints motivate the use of specific data structures and a particular partitioning of the input for efficient parallel processing.

Our Marching Cubes miniapp provides some generally useful implementations of the algorithm and allows users to easily modify them to fit new constraints. It includes a serial implementation of the algorithm, a shared memory implementation using OpenMP, a distributed memory implementation using MPI, and a combination of the latter two. Users can add functionality or optimize the miniapp depending on their needs relatively easily compared to working on software packages like VTK.

Although Marching Cubes is a well known algorithm, we review its specifics in Section 2 for users, especially those generally interested in Mantevo miniapps, who may not be visualization specialists. The next section describes the overall structure of this project's source code, and it helps users compile and run our software. Section 4 covers the most important implementation details for those who wish to modify or extend our code.

2. Marching Cubes. Marching Cubes starts with a user specified isovalue. The goal is to produce a three dimensional mesh contouring the isovalue in a given image volume. Mathematically, an image volume is a discrete scalar field with each point in space bearing an image value. To produce this contour mesh, we start at the first cell of the image volume and move (march) from cell to cell (cube to cube) and compare the vertex values to the isovalue. If a pair of vertex values straddle the isovalue, we interpolate the mesh intersection along the edge joining these two vertices.

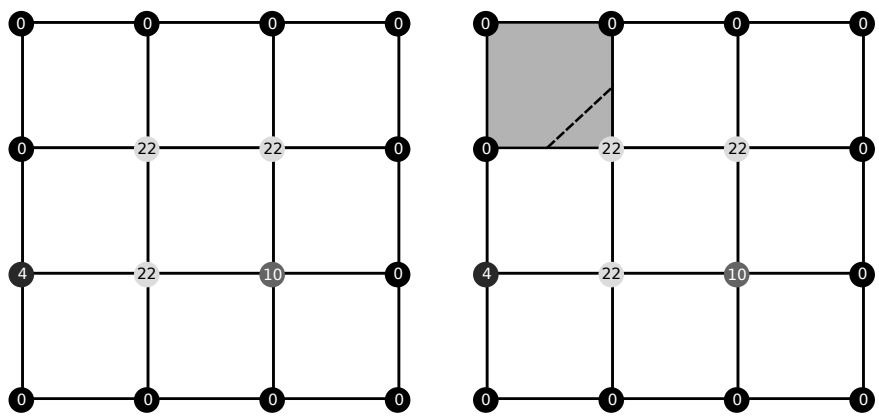
After finding the mesh intersection points we group them into triangles. Triangles are the geometrical unit that make up three dimensional surfaces in computer graphics. The sets of triangles from each cell, in the end, form the final isosurface mesh desired.

Sections 2.1 and 2.2 review the essential procedure Cline *et al.* [6] outline in their original Marching Cubes paper, or Newman *et al.* [7] surveyed more recently. Section 2.3

*Univeristy of California Santa Barbara, sjmunn@umail.ucsb.edu
†Sandia National Laboratories, kmorel@sandia.gov

delves into parallelization difficulties, which are typically in resolved in implementations of Marching Cubes but left implicit in the literature.

2.1. Marching Squares. To better illustrate the process, we consider the marching squares algorithm. This is the same idea as Marching Cubes except that the input data is a two dimensional image and the output is a contour line rather than a surface. In Figure 2.1(a) we present an example input image. The numbers written at the vertices represent the image values at those points. Figure 2.1(b) shows the first step of marching squares in which we interpolate the contour line between the vertex values of the grid.



(a) Initial image (b) First step

Fig. 2.1: Initial image and first step of marching squares.

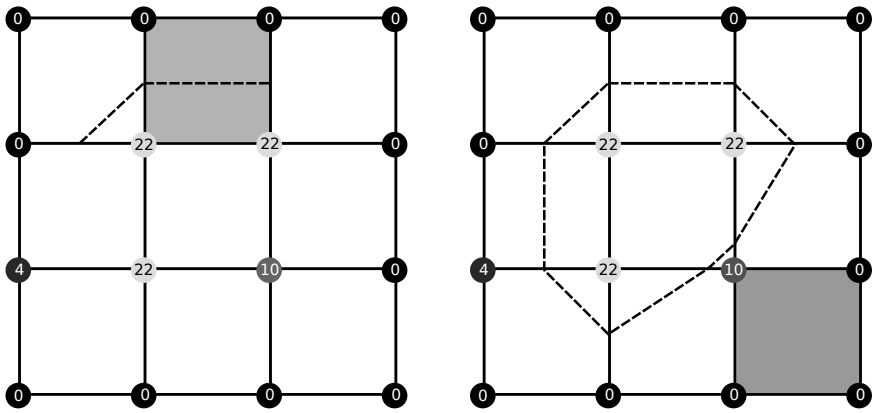
After computing the intersection points specifying the line in the first square we move to the second in Figure 2.2(a). Then we proceed through each square in the image file until we end at the final step in Figure 2.2(b) with our complete contour line.

For marching squares, there are $2^4 = 16$ possible combinations of vertex indices above or below the isovalue. For each case, we know which edges the isoline will intersect, and we can interpolate the intersection point along that edge.

2.2. Building the Isosurface Mesh. For Marching Cubes, there are $2^8 = 256$ possible combinations; however, it is most efficient to implement a lookup table mapping the cases (for example vertex 3 and 4 are above the isovalue) to edges where the contour mesh will intersect (edges 2, 3, 4, 7, 8, and 10 in the previous case). Figure 2.3(a) contains an example case using the vertex indexing convention we use in the software. Once we know the vertices that are above and below the isovalue, we look up the corresponding edges where the mesh will intersect.

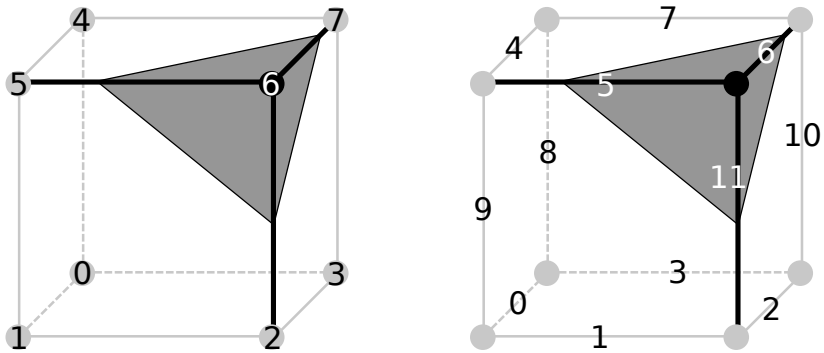
In practice, this map returns intersection edges for each triangle to be added to the isosurface mesh. For example, if there are four intersection points on the cell, our map will return 6 edges relevant to the 6 points definining the two triangles, which we will add to the output mesh.

Adding triangles to the mesh one after the other poses a problem though. Adjacent cubes will share edges and the triangles within these cubes will share points at the same



(a) Second step (b) Final step

Fig. 2.2: Second and finals steps of marching squares.



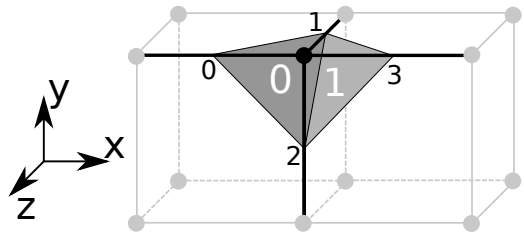
(a) Vertex indices (b) Edge indices

Fig. 2.3: Cube of data: (a) Only vertex number 6 has a value beneath the isovalue, (b) The contour mesh intersect edges 5,6, and 11.

coordinate; therefore, consistently saving the coordinates for all three points in each triangle within a cube will result in redundancy.

The solution is to store point coordinates in a separate data structure from the mesh. Point coordinates are kept in an array whereas the mesh is kept as a collection of point indices referring to particular coordinates. Figure 2.4 illustrates this principle.

There are several methods to determine if an intersection point already exists in memory. A popular technique is to partition the image volume into blocks. A data structure is then built to keep track of points in each block. When marching inside a block, this data structure



Mesh	triangle0	triangle1
	0	1
	1	2
	2	3

Point Index	0	1	2	3
x	0.5	1	1	1.5
y	1	1	0.5	1
z	1	0.5	1	1

Fig. 2.4: Illustration of the mesh data structure. The mesh is a collection of triangles. Triangles are triplets of point indices.

returns the nearby points to search for redundancies.

For this project we chose a different approach for performance reasons. Searching even a small set of floating point numbers can be time consuming, so we devise a method that does not require any searching.

To begin, we establish a consistent convention for indexing each edge in the data set (see Section 4.3). When a new intersection point is located, we add the point index to a data structure mapping edge indices to point indices. If we locate an intersection point where the edge already has a point index associated with it, then we use the preexisting point index rather than create a new one.

2.3. Parallelizing Marching Cubes. To make full use of modern, multicore machines, Marching Cubes must be done in parallel. There are two major types of parallelism this miniapp leverages: shared memory and distributed memory.

2.3.1. Shared memory. In a shared memory setting, threads all have access to the entire image volume. Before marching through the data, we partition the space into small blocks. During run-time, threads accept ranges within these blocks over which to march and add triangles to the mesh.

Our previous strategy of mainting an edge to point index map for eliminating redudant points is problematic here though. Concurrent threads cannot simultaneously write to the map as this would cause a race condition.

Miller *et al.*'s solution [8] is to assign a private map and mesh to each thread. These thread maps will only contain unique points thanks to their corresponding thread map.

Once all the threads have marched through all their assigned blocks, Miller *et al.* propose we merge all the thread meshes into a fully connected final mesh. This step is necessary because shared memory threads work on small data blocks and would result in small mesh fragments otherwise. Section 4.4 covers the specifics of this merging process.

2.3.2. Distributed Memory. For distributed memory implementations, a master process assigns data block ranges to concurrent processes. Each process loads only the block of data it has been assigned to its local memory from the original data file. Since distributed memory processes work on larger data blocks than their shared memory counterparts, it is not necessary to merge meshes from the different processes together.

More specifically, the reason why we do not merge meshes from the different processes is because in distributed memory it is more efficient to maintain redundancy at the data boundaries. This redundancy would bloat a thread's memory; however, for processes it is avoids lengthy communications with other concurrent processes.

3. Software Design Choices. Our project aims to provide a set of basis implementations easily combined or adjusted by the user. Each implementation leverages the advantages of typical hardware architectures found in industry. Because the various implementations may not all be relevant to every user, we separate them into different folders with their own compiling process. This way, users do not need to resolve all the dependencies for each implementation; they can pick the relevant ones for their use case and ignore the rest.

To this effect, we maintain a *common* folder with general purpose, implementation independent code. The other directories all have their own *main.cpp* and *Makefile* files for compiling and running Marching Cubes.

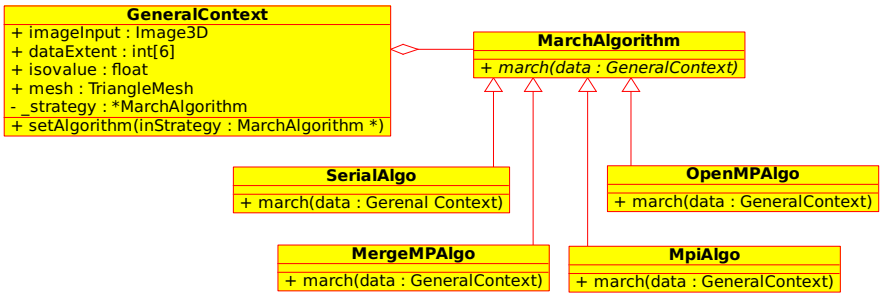


Fig. 3.1: The Strategy design pattern: `GeneralContext` includes a member pointer to an object inheriting from the `MarchAlgorithm` class.

For clarity, we follow the Strategy design pattern [3] in which individual implementations appear as distinct objects inheriting from an abstract class representing their common goal. Figure 3.1 illustrates this structure in a UML diagram.

Each implementation directory contains the definition for a child class of the *MarchAlgorithm* parent. This definition's header file encapsulates all the implementation's specific dependencies, neatly confining them within their implementation's folder.

To compile an executable implementation, users run the build process in the *common* directory first. Here, the compiler will not need any libraries outside of standard C++. Depending on the support users have, they may proceed to run the build process in the implementation directories.

4. Implementation Fine Points. Writing any efficient implementation of Marching Cubes in C++ requires a careful study of the data structures and subroutines employed. This section collects the important, and perhaps less intuitive, particulars of the miniapp. The benefit will especially be for readers who wish to modify the miniapp's code.

4.1. File In/Out Operations. Since Mantevo miniapps are meant to be minimal, we provide support only for one kind of file: vtk legacy. For image volumes, this consists of a

header containing information such as the file dimensions, spacing, and origin in 3-d space. The point values follow the header and are stored in binary. The file stores points along the x-axis first, then the y and z directions. Thus a point with coordinates $\mathbf{r} = \begin{pmatrix} x & y & z \end{pmatrix}$ will have a position in the file given by,

$$p = x + R_x \times y + R_x \cdot R_y \times z \tag{4.1}$$

where the coordinates start at zero and R_x is the image volume range along the x-axis [4].

During runtime the software stores image data in memory in the exact same order, so Equation 4.1 is relevant for finding data values during execution as well.

In practice, the *LoadImage3D* class loads the input file; however, we split this into a two step procedure because the distrubuted memory implementation requires multiple file reading objects for each process. In the *main.cpp* file for any implementation, the *LoadImage3D* object reads only the header for a given file. Execution then passes the C++ object by reference to its algorithm object, which decides to read the entire file or only a given block of data.

4.2. Triangle Mesh Class. The triangle mesh encapsulates three private member arrays: points, normals, and indices. Each of these is an array of Triplets–a template class object with three private data members. Point coordinates and normal vector directions correspond to float type Triplet objects and sets of triangle indices correspond to integer type Triplet objects.

Users may merge triangle meshes from parallel threads together without checking for point redundancy using the + = operator; however, this is not thread safe and must be done in an OpenMP critical directive. The + = operator's effect amounts to concatenating the private member arrays after incrementing the point indices of the right-hand value by the number of points in the left-hand Triangle Mesh.

Merging meshes while eliminating redundancies in the points to produce a fully connected mesh is a different matter, which involves the *DuplicateRemover* class described in Section 4.4. The *DuplicateRemover* class depends on a consistent convention for indexing edges, which we describe in the next section.

4.3. Edge indexing. The purpose of indexing the edges in the image volume is to uniquely associate interpolated points with the edge on which they belong. Each edge should host only one point. When a thread or process is marching through its block of data, it uses the edge indices to avoid adding redundant points to its own *TriangleMesh* object. Later, as we merge meshes from different threads, the edge indices help identify redundant points between different *TriangleMesh* objects through a process detailed in Section 4.4.

Edges follow essentially the same indexing convention as data points in the file. We start with edges along the x-axis and proceed to the y and z axes. Figure 4.1 lays out the edge numbers along the x and y axes, while Figure 4.2 demonstrates numbering on the z-axis edges.

How do we determine the edge number in the image file given the Cartesian coordinates of a cell and the local edge number (between 0 and 11)?

Considering the local edge number convention from Figure 2.3(b), we denote $E(a)$ the global edge number given a local edge identifier of a. Further noting $R_x = d_x - 1$ where d_x is the data dimension along the x-axis we have,

$$\begin{aligned}
E(0) &= x + R_x y + (R_y + 1) R_x z \\
E(3) &= N_x + x + (R_x + 1) y + (R_x + 1) R_y z \\
E(8) &= N_{xy} + x + (R_x + 1) y + (R_x + 1)(R_y + 1) z
\end{aligned}$$

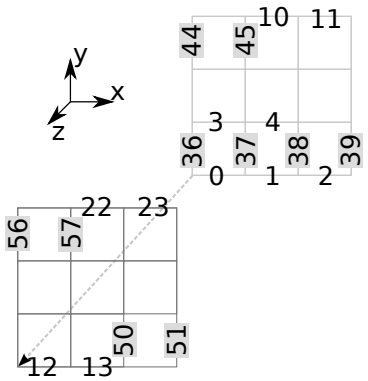


Fig. 4.1: Edge numbers along the x and y axes.

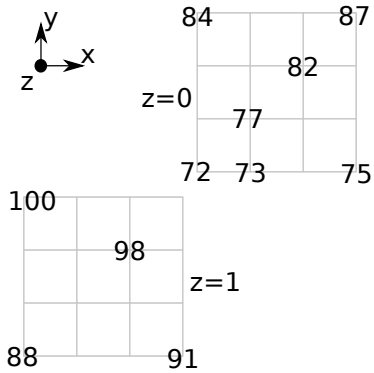


Fig. 4.2: Edge numbers along the x and y axes.

where $N_x = R_x\,(R_y + 1)\,(R_z + 1)$ and $N_{xy} = N_x + (R_x + 1)\,R_y\,(R_z + 1)$.

Offsetting the x, y, or z values in the above equations will allow us to find the global edge indices for other local indices as such,

$$
\begin{aligned}
E\,(1, x, y, z) &= E\,(3, x + 1, y, z) \\
E\,(2, x, y, z) &= E\,(0, x, y + 1, z) \\
E\,(4, x, y, z) &= E\,(0, x, y, z + 1) \\
E\,(5, x, y, z) &= E\,(3, x + 1, y, z + 1)
\end{aligned}
$$

We do not cover all the cases here to avoid being pedantic. Interested readers can examine the *EdgeIndexer.cpp* file in the *common/Algorithm* directory of the miniapp.

In the general software context, an algorithm object (inheriting from *MarchAlgorithm*) will instantiate an *EdgeIndexer* object for the data block it is assigned to. Threads, in a shared memory context, use a global edge indexing scheme, whereas processes use a local scheme since we will not merge their output meshes. The *EdgeIndexer* class only implements the above equations. It is the DuplicateRemover class that removes redundant points.

4.4. Duplicate Remover and Point Map classes. The duplicate remover encapsulates an array of *edgePointPair* structures. The latter's members are: an edge index number, a point index number, and a new point index number for removing duplicates. The purpose of this object, based on Miller *et al.*'s paper [8], is to find duplicate points between two meshes, create new point indices for all the unique points, and map old indices to new ones. The *buildMesh()* function will use this map to create a new, fully connected *TraingleMesh* object.

Edge number	20	22	20	24
Point number	0	1	2	3

Table 4.1: An example initial state for the the duplicate remover's data array.

The duplicate remover's data array begins with an array of unsorted *edgePointPair* structures. Table 4.1 exemplifies an initial state for the object. The edge and point numbers simply appear in the order they emerged during the Marching Cubes process.

Edge number	20	20	22	24
Point number	0	2	1	3

Table 4.2: Sorted duplicate remover data array.

To merge two meshes, the user first calls a member function in the duplicate remover to sort itself by edge number. For the data in Table 4.1, table 4.2 represents the sorted output of this member function.

Any point on the same edge will coincide and is thus a duplicate. A member function of the duplicate remover will iterate through its data array comparing each edge to the previous one while writing new point indices to the data array. New point indices start at zero and are only incremented when a new edge appears. Table 4.3 shows the results of such an iteration.

Edge number	20	20	22	24
Point number	0	2	1	3
New Point Number	0	0	1	2

Table 4.3: Sorted duplicate remover data array, with new point indices added.

With the new unique point identifiers in memory, the duplicate remover proceeds to map old to new point indices. Algorithm 15 details the steps for producing this map.

Finally, the *buildMesh()* function accepts this map and builds a whole new mesh from the one it is given where redundant indices in the triangles are replaced with a unique identifier.

4.5. Timing and Reporting. Reporting performance measures, relevant run time data, and errors is vital for a miniapp. We provide three types of reporting in our software: runtime reporting via a *Log* class that reports data as it is being processed, error reporting

Algorithm 15 Map old point identifiers to new ones

```
Map is an array
for iDataPt ∈ |DataArray| do
    Map[DataArray[iDataPt].oldPointNumber]
        = DataArray[iDataPt].newPointNumber
```

via the C++ standard library, and a document class that saves performance data in a YAML file for comparing different runs.

The Log class provides a performance aware method for displaying runtime data. The user must hard code its reporting level into the main.cpp file before compiling. This means that ignored messages are never compiled and thus do not impact performance in any way.

4.6. Parallel Processing Blocks. The *Ranges* object stores the extent of a block of data sent to a thread or process. In OpenMP, these blocks are cubic chunks of data, significantly smaller than the entire data size for load balancing purposes. In MPI, these are slabs covering the full x and y extent but partitioned in the z direction by the number of available processes.

5. Conclusion. Marching Cubes is an essential tool for scientific visualization and its many subroutines are representative of the general tasks performed in the field. Systems able to perform Marching Cubes in a timely manner will also be effective for a host of other visualization techniques.

Our project's purpose is twofold: testing new hardware and testing implementation algorithms tailored to new hardware. We achieve this dual purpose by reducing the Marching Cubes algorithm strictly to its computationally relevant components. To address the variety of hardware available, however, we also provide several basic implementations of the same overall procedure. Users can adapt the most relevant implementation for their own usage.

In the future we hope our miniapp will provide a massively parallel implementation of Marching Cubes although at the time of writing, it does not. GPU's or the new Intel Xeon Phi coprocessors would leverage this to achieve run times currently inaccessible to the project. The major algorithmic difference, inspired by Lo *et al.* [5], would involve counting all the intersection points before computing their location so as to store points during the marching process in shared rather than distributed memory locations.

REFERENCES

[1] *Home of the mantevo project.* Accessed: 2015-09-08.
[2] U. Ayachit, *ParaView Guide*, Kitware Inc., 2015.
[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-oriented Software*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.
[4] K. Inc., *VTK User's Guide*, Kitware Inc., 2010.
[5] L.-t. Lo, C. Sewell, and J. P. Ahrens, *Piston: A portable cross-platform framework for data-parallel visualization operators.*, in EGPGV, 2012, pp. 11–20.
[6] W. E. Lorensen and H. E. Cline, *Marching cubes: A high resolution 3d surface construction algorithm*, SIGGRAPH Comput. Graph., 21 (1987), pp. 163–169.
[7] W. E. Lorensen and H. E. Cline, *Marching cubes: A high resolution 3d surface construction algorithm*, in ACM siggraph computer graphics, vol. 21, ACM, 1987, pp. 163–169.
[8] R. Miller, K. Moreland, and K.-L. Ma, *Finely-Threaded History-Based Topology Computation*, in Eurographics Symposium on Parallel Graphics and Visualization, M. Amor and M. Hadwiger, eds., The Eurographics Association, 2014.

COMPARING POWER PROFILES OF MOLECULAR DYNAMICS SIMULATORS

JORDAN T. RAITSES* AND RYAN E. GRANT†

Abstract. The increasing power demands of High Performance Computing (HPC) clusters has led to the creation of component-level power measurement tools such as Power Insight and Intel's RAPL. To interface with these hardware-level power measurement tools, a portable library known as Power API has been developed at Sandia National Laboratories. Now that the research community can access detailed power information from HPCs, they can develop software (even to the kernel level) that intelligently monitors and adjusts its own power usage to meet efficiency or performance goals. Sandia uses export controlled or highly technical software to benchmark its systems, but there are non-export controlled, simpler analogs known as mini apps that are made publicly available for collaborative research purposes. These mini apps approximate the performance of the full software while being much easier to understand and not posing an export risk. This study shows that the molecular dynamics mini app, MiniMD, approximates the power profile of its full-size equivalent, LAMMPS, under similar input "decks".[1]

1. Introduction. The number of transistors on chips has doubled every two years since the mid 1970's which results in a doubling of power density on these chips (though this trend is beginning to slow, the salient effects are still relevant). High Performance Computing (HPC) clusters are composed of thousands to millions of processors and thus require large and ever-increasing amounts of power. As a result, power/performance is an important benchmark for operators of HPC clusters. Power Insight is a device, developed in tandem with Sandia National Laboratories and Penguin Computing, which allows per-component power measurement in an HPC node. The PowerAPI was also developed at Sandia as a portable software library for interfacing with Power Insight or other power measurement tools (e.g. RAPL). Such software and hardware allows these systems to be "aware" of their own energy usage so that specialized software (or even Operating Systems) can run more efficiently.

The many cores of a HPC make them well suited for solving problems with many data points. Such problems, like molecular dynamics simulations, are split into smaller problems, each of which is handled by a different node. LAMMPS is a program for modeling these molecular dynamics problems on HPC clusters and can run simulations on hundreds of machines simultaneously. While it is intended to be used for these simulations, it is also useful for the HPC research community as a benchmark of a HPC cluster's performance. However, it is thousands of lines long and highly technical so Computer Science researchers who do not understand molecular dynamics, cannot easily use or understand it. MiniMD was built as an analog to LAMMPS, performing the same core functions without being as unwieldy for non-experts. While its performance matches that of LAMMPS, it is unknown whether or not it has a similar power profile–an aspect that is of importance if one is doing power research.

2. Method. In this study, a script was used to run the PowerAPI on a node of one of the HPC test beds at Sandia Labs. For these tests, one node of Shepard with 36 Dual Intel Xeon Haswell cores and 128 GB DDR4 RAM and Power Insight v2 was used. The script initialized data collection through the PowerAPI on the appropriate node, waited one second (to allow collection of background "noise"), and ran the target program (LAMMPS or MiniMD). The data from each run was collated by means of a python script which added data from each new run to a database, organizing the power readings by the timestamp at

*Binghamton University, jraitse1@binghamton.edu
†Sandia National Laboratories, regrant@sandia.gov

which each reading was gathered. This database was used to find median power readings across all program runs.

3. Results. No formal data analysis (e.g. calculation of standard deviation) has yet been run. However the graphs seem to indicate that, for the duration of the programs' run-times, their power profiles are very similar (as shown in Figure 3.1).

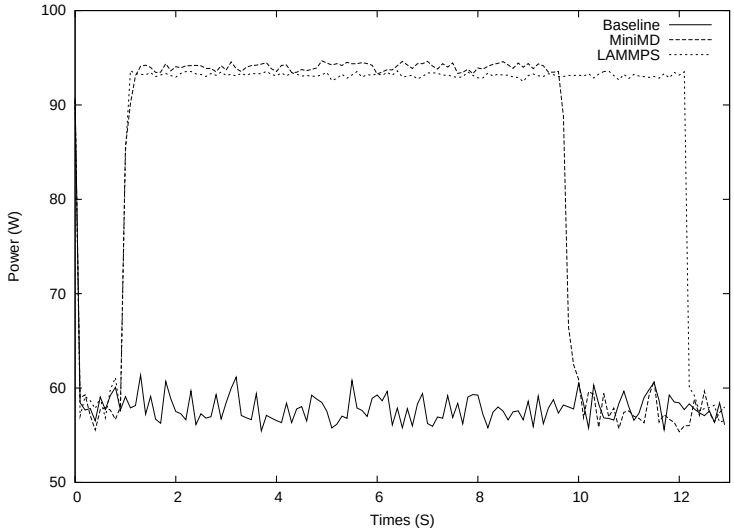


Fig. 3.1: LAMMPS and MiniMD Power Measurements over Runtime

This is compared to a separate test in which LAMMPS alone was run on several input decks. In the latter test, it is clear to see where each test ends and that the power profiles of the program's different runs are completely different (see Figure 3.2).

These functions show a sharp, initial increase in power consumption as the system goes from resting (indicated by "Baseline") to full power. Upon reaching full power, the lines plateau, indicating maximum power consumption for the system. Finally, the lines quickly descend back to the baseline as the programs finish running. This rectangular function illustrates that the programs in my test, once started, largely ran at full power until they completed. Other programs may have cyclical or curved functions of power draw, corresponding to 'busy' and 'waiting' sections, but I haven't observed any in my tests. It is also possible that the power readings (taken at 1 Hz) were not sufficiently granular to show such fluctuations.

REFERENCES

[1] J. H. Laros, P. Pokorny, and D. DeBonis, *Powerinsight-a commodity power measurement capability*, (2013), pp. 1–6.

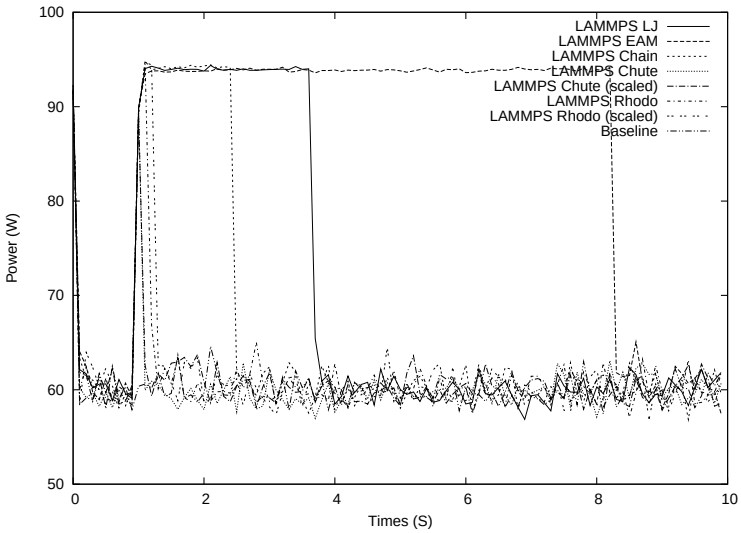


Fig. 3.2: Various LAMMPS Power Measurements over Runtime

DISPLAYING MATERIAL PROPERTIES WITH EOS TABLE VIEWER

JARED M. STATEN*, JOHN H. CARPENTER†, AND ALLEN C. ROBINSON‡

Abstract. Equations of state represent the equilibrium thermodynamic state properties such as pressure or sound speed as a function of two other variables such as density and energy. For a given material, equation of state (EOS) tables can be used to quickly calculate the properties and the state or phase the material will be at each point of phase space. It can be difficult and expensive to subject materials to extreme conditions and one must still infer properties from limited external measurements. Computer modeling is essential to study the properties of materials over a wide range of phase space. A good modeling code can save valuable time and money. However, good tools are needed to study carefully the EOS closure surfaces and models of experiments which place material on certain trajectories in phase space. This paper documents efforts to build a graphical user interface intended to be helpful in understanding improvements in accuracy, usability, and functionality for such modeling efforts. The ultimate product is a fully-functional, visually-pleasing EOS table visualization program.

1. Introduction. When running computational simulations, it is critical to know how a material will act when subjected to varying temperature, pressure, density, and other conditions. For example, the Euler equations, shown below, which conserve mass, momentum and energy, can be used to model material properties at varying conditions.

$$\dot{\rho} + \rho \nabla \cdot \mathbf{u} = 0 \tag{1.1}$$

$$\rho \dot{\mathbf{u}} + \nabla p = 0 \tag{1.2}$$

$$\rho \dot{e} + p \nabla \cdot \mathbf{u} = 0 \tag{1.3}$$

The 'dot' notation refers to the total material derivative. Note that there are 5 equations (mass, momentum, energy) and 6 unknowns (density, velocity, energy, pressure) in three dimensions. In order to solve these equations we find a way to reduce the number of unknowns and 'close' the system. It is very common to assume that we can compute the pressure from equilibrium thermodynamics assumptions. This closure property, called an equation of state (EOS), is highly specific to the material at hand, and essentially requires that pressure be given as a function of density and energy. The closure property for aluminum, for example, is quite different from the closure property for an ideal gas. An ideal gas EOS is very simple, and is displayed below as an example of a possible closure property for Euler's equations. These Euler equations are the fundamental equations representing continuum mechanics in compressible media. Extentions of these equations add additional physics and additional closure relations but the basic concepts are the same.

$$p = \rho(\gamma - 1)e \tag{1.4}$$

Material properties can be accurately modelled with the use of these equations, but it can be difficult to picture what is going on and fully understand the meaning of the closures and how they relate to the dynamics of an experimental configurations. For this reason, a computer program to visualize equations of state is incredibly valuable. Such a tool allows

*La Cueva High School, Brigham Young University (Fall 2017), jared.staten@yahoo.com
†Sandia National Laboratories, jhcarpe@sandia.gov
‡Sandia National Laboratories, acrobin@sandia.gov

users to view visual representations of the equations of state for a material and to place their calculations on this surface. This paper documents efforts to build an effective EOS viewer in the SHIVR tool for analyzing results from hydrodynamic simulation software. Such codes are sometimes called "hydrocodes" in the DOE community. The SHIVR tool was initially created as part of the ALEGRA project at Sandia [1]. A great deal of effort can go into building effective simulation tools for production usage, verifying and evaluating the numerical methods, proposing better approaches and models and evaluating simulations relative to experiments. Simulation tools are critical to help design experiments. The better the tool suite the more likely that insight can be obtained from the actual physical experiment. This project is part of a general effort to improve the user tools suite.

2. Building an effective EOS table viewer. This project began with an initial prototype EOS table viewer created by a student, Brian Kelley, during the summer of 2014, to build a tool to visualize the UTri (Unstructured Triangluar) equation of state table format recently proposed and supported at Sandia. This format is based on a triagular grid, is double noded at phase lines to allow for discontinuities in the phase surface and also supports uncertainty quantification via a modal expansion assumption. Our purpose has been to make the existing program more robust, remove bugs, and add features to the point where the viewer could be generally useful to both equations of state developers and simulation software users utilizing the UTri format.

2.1. User interface organization. The user interface consists of two panes, which serve two different purposes, as indicated by their names. The two panes are depicted in Figure 2.1. The Background pane controls the table and variable displayed, the location of the contours, and the specifics of the special contour and clicked point. The Display pane primarily controls the color of parts of the display and whether or not they are present.

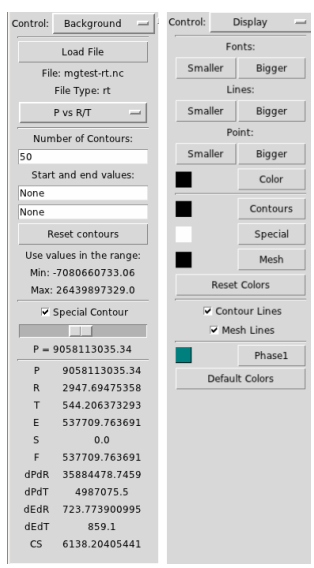


Fig. 2.1: The Background pane (left) controls the location of displayed objects and gives table information. The Display pane (right) controls size and color of displayed objects as well as whether they are displayed.

2.2. Continuous contours and full triangle shading. The old sort method for contour points was faulty, and resulted in discontinuous contours. The first focus of this project was to fix the contour point sorting process to produce continuous contours. Figure 2.2 shows the effect of the old method and the new method on contour display.

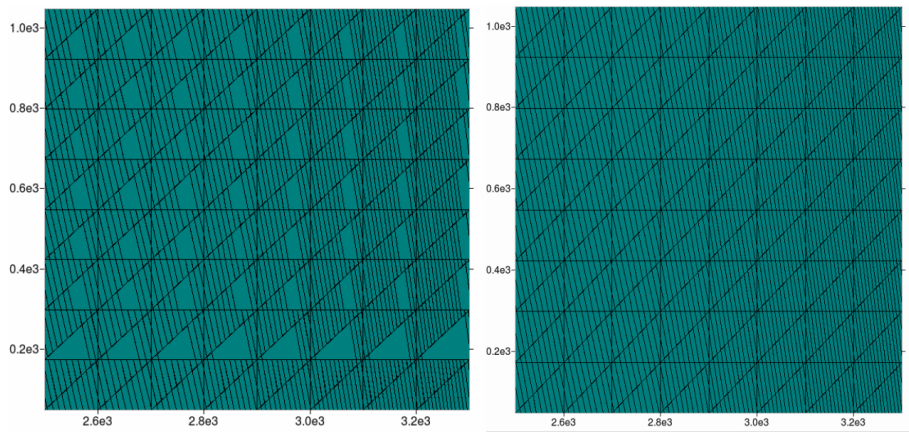


Fig. 2.2: Discontinuous contours with the old method (left) and continuous contours with the new method (right)

An additional issue with the logic of the old program caused triangles only to be shaded if they had a node present on the screen. This was causing some regions to be unshaded on the display because the nodes were not visible, as indicated in Figure 2.3.

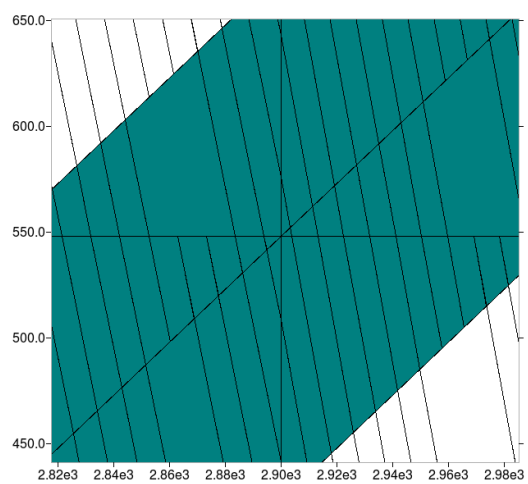


Fig. 2.3: Triangles without a node present on the screen were formerly left uncolored.

2.3. Point marking last user click. When a user click occurs, the Background window displays the information for the clicked point. If the user moves the mouse, however,

it is easy to lose track of the clicked location. Now, a point is drawn at the location of the user click to indicate the location of the last user click and give meaning to the displayed information. This functionality is shown in Figure 2.4.

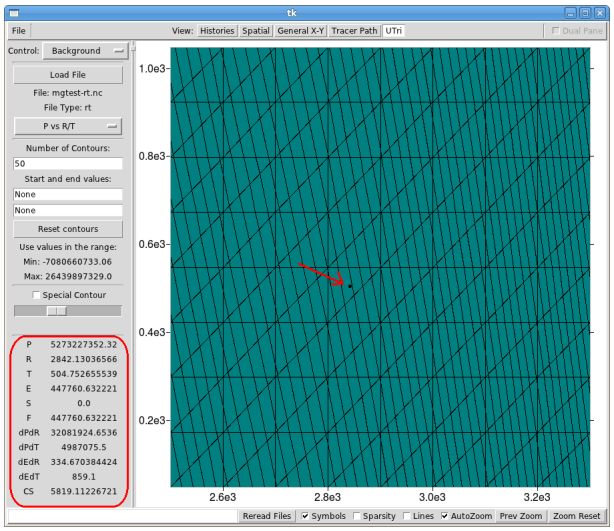


Fig. 2.4: The information displayed on the Background pane describes the values of the clicked location, which is indicated by a small point.

Widgets were added to allow the user to change the color and size of the point, if desired.

2.4. User-specified contour display. An additional focus of the project was to allow users to specify more information than before. The Background pane allows users to enter min and max values for the contours displayed, as well as the number of contours. The entry widgets and effect of user input are shown in Figure 2.5. Contour values can also be reset to the default values. To prevent users from entering invalid values, the dependent variable max and min are displayed to give the user an idea of a reasonable range of values.

2.5. Special contour. One of the biggest changes in the new table display program is the use of a special contour. The special contour shows the dependent variable value contour line through the point clicked by the user. The special contour can, of course, be turned on and off with a button on the user interface, as well as with the spacebar.

In addition to appearing through clicked points, the special contour can be moved with a slider on the background pane. Figure 2.6 shows the special contour and corresponding slider. The slider shows the proportion between the min and max represented by the special contour. For instance, if the special contour represents the value exactly between the min and max visible, as represented in the Figure 2.6, the slider shows a proportion of 0.5. User manipulation of the slider can move the special contour across the screen, toward the visible min and max, accordingly. Users can also move the slider and contour with the arrow keys, and change the special contour color if desired.

When zoom conditions are changed, the slider detects the visible min and max dependent variable values and resets accordingly. This way, the proportion displayed by the slider

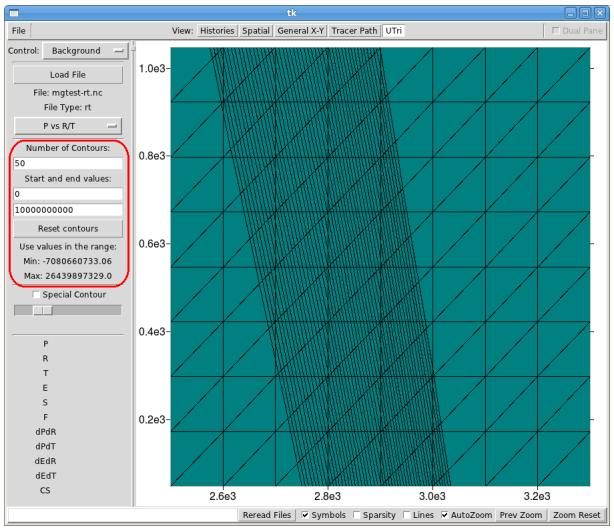


Fig. 2.5: When specified, contours are only displayed between the min and max set by the user.

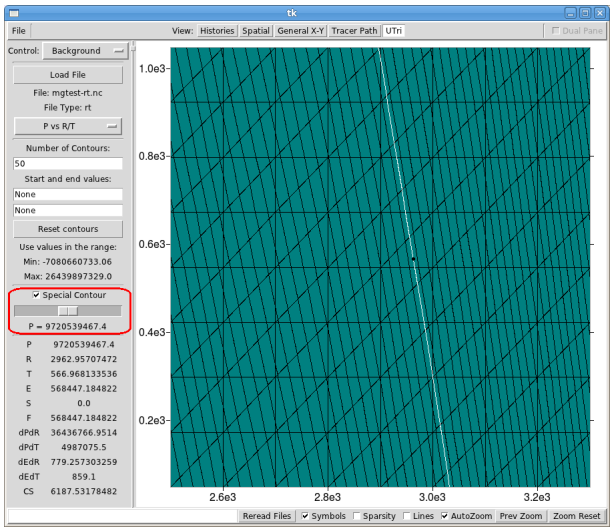


Fig. 2.6: The slider reflects the location of the special contour. In this case, the slider has a value of 0.5, meaning the special contour is directly between the min and max.

always represents the location of the special contour relative to the visible min and max dependent variable values.

2.6. Phase color editing when phase is present. It is critical that users are able to distinguish between phases on the table. When a table is read into the display program, each phase present in the table is assigned a color such that every phase is easily distinguishable from the others. If desired, however, users can specify colors for each phase value. It is

critical that users can only change the color of phases that are actively displayed on the screen. The phase color modification widgets are shown in Figure 2.7.

2.7. Log scale. For some tables, it is necessary to use log scale to display all of the important aspects of the table. The pre existing external log scale buttons were not functional, so they were removed. The updated log scale uses no widgets on the user interface. Log scale is now used for the tables that needed it by changing independent variable values to the log of the value when the table is read in. The values go through calculation as logs, and then every value that is displayed externally is converted from log back to the real value. For axis labels, the labels are still even spatially, but they do not represent consistent numeric intervals. This illustrates to the user that log scaling is being used. Figure 2.7 shows a table that uses log scale to display critical values.

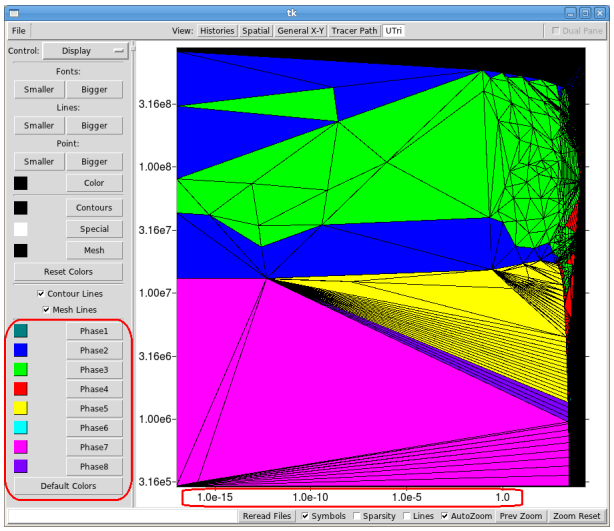


Fig. 2.7: Phase color modification widgets appear for every phase active in the display. Also, this table uses log scale to show more detail at low values.

3. User Reception. We expect users to be excited and intrigued when this new viewer capability becomes available. We believe that the clear representation of phase regions is likely to be most helpful in understanding simulations and in developing a positive feedback between continuum modeling and equation of state modeling. The first production UTri EOS table will be an Aluminum table. Additional developments for materials and other closure properties such as electrical conductivity will add even more value.

4. Conclusions. Due to the expense and complications of physically subjecting materials to extreme conditions including pressure and temperature, computer simulations and visualizations are extremely valuable tools to determine the phase of the material and how this state relates to the physical simulation at hand through continuum equation modeling. The purpose of this project was to build a functional and accurate tool for visualizing the UTri equation of state tables and thus provide value in important scientific and engineering efforts.

REFERENCES

[1] A. C. Robinson et al., *ALEGRA: An arbitrary Lagrangian-Eulerian multimaterial, multiphysics code*, in Proceedings of the 46th AIAA Aerospace Sciences Meeting, Reno, NV, January 2008. AIAA-2008-1235.